Electronic Notes in Theoretical Computer Science 60 (2003)
http://www.elsevier.nl/locate/entcs/volume60.html 405 pages

Types for Modules

Claudio V. Russo

The University of Edinburgh, Scotland

Keywords: existential types, generativity, modules, polymorphism, programming languages, subtyping, Standard ML, type theory

Preface

This monograph is derived, with only minor typographical revisions, from my University of Edinburgh doctoral thesis [Rus98b], examined in 1998. For better or worse, I decided to leave the main body of the work unchanged and to exclude any account of subsequent related research. However, I have added an epilogue, not contained in [Rus98b], that relates the thesis to my more recent work on elaborating and implementing my proposed extensions to Modules in Moscow ML [RRS00], a widely used Standard ML compiler. I hope the reader will find this additional material of interest too.

Claudio V. Russo
Microsoft Research Ltd,
7 J J Thomson Ave,
Cambridge CB3 0FB, UK

October 2003

Abstract

The programming language Standard ML is an amalgam of two, largely orthogonal, languages. The Core language expresses details of algorithms and data structures. The Modules language expresses the modular architecture of a software system. Both languages are statically typed, with their static and dynamic semantics specified by a formal definition.

Over the past decade, Standard ML Modules has been the source of inspiration for much research into the type-theoretic foundations of modules languages. Despite these efforts, a proper type-theoretic understanding of its static semantics has remained elusive. In this thesis, we use Type Theory as a guideline to reformulate the unconventional static semantics of Modules, providing a basis for useful extensions to the Modules language.

Our starting point is a stylised presentation of the existing static semantics of Modules, parameterised by an arbitrary Core language. We claim that the type-theoretic concepts underlying Modules are type parameterisation, type quantification and subtyping. We substantiate this claim by giving a provably equivalent semantics with an alternative, more type-theoretic presentation. In particular, we show that the notion of type generativity corresponds to existential quantification over types. In contrast to previous accounts, our analysis does not involve first-order dependent types.

Our first extension generalises Modules to higher-order, allowing modules to take parameterised modules as arguments, and return them as results. We go beyond previous proposals for higher-order Modules by supporting a notion of type generativity. We give a sound and complete algorithm for type-checking higher-order Modules. Our second extension permits modules to be treated as first-class citizens of an ML-like Core language, greatly extending the range of computations on modules. Each extension arises from a natural generalisation of our type-theoretic semantics.

This thesis also addresses two pragmatic concerns. First, we propose a simple approach to the separate compilation of Modules, which is adequate in practice but has theoretical limitations. We suggest a modified syntax and semantics that alleviates these limitations. Second, we study the type inference problem posed by uniting our extensions to higher-order and first-class modules with an implicitly-typed, ML-like Core language. We present a hybrid type inference algorithm that integrates the classical algorithm for ML with the type-checking algorithm for Modules.

Acknowledgements

I would like to thank Donald Sannella, my supervisor, for taking me on as a student and guiding my work throughout the years.

Healfdene Goguen, my friend and second supervisor, deserves special thanks for stepping in when the chips were down and prodding me to continue and finally finish my work. His weekly supervision helped me shape my jumbled ideas from little more than code into their current form.

Thanks to Stefan Kahrs, James McKinna and, in particular, David Aspinall for their technical feedback.

Thanks to Stuart Anderson and Donald MacKenzie for giving me the opportunity to explore Computer Science beyond the horizons of my thesis.

My parents supported me both financially and with encouragement. None of this work would have been possible without them.

Patricia, now my wife, was always there to pick up the pieces. Much of my time was also hers, and I am indebted to her for all her support.

My close friends Miri, Pete & Tim, Simon, David (again), Adriana, Matt, Gordo, Jen, and Jo offered invaluable distractions.

I am grateful to have been partially supported by an award from the U.K. Science and Engineering Research Council.

I would also like to thank Don Sannella, Peter Sestoft and Ken Friis-Larsen for the fun I have had extending Moscow ML.

Contents

Chapter 1

Introduction

Standard ML is a very high level programming language that is suited for the construction of both small and large programs. It is actually an amalgam of two, to a large extent orthogonal, programming languages. "Programming in the small" is captured by the *Core* language. "Programming in the large" is captured by the *Modules* language, which provides constructs for organising related Core language definitions into self-contained modules with descriptive interfaces. While the Core is used to express details of algorithms and data structures, Modules is used to express the overall *architecture* of a software system.

Modules allows definitions of identifiers denoting Core language types and terms to be packaged together into possibly nested *structures*. Access to structure components is by the dot notation and provides good control of the name space in a large program development. Structures are *transparent*: by default, the identity of a type component within a structure is evident from outside the structure. This provides good support for the incremental construction of large programs.

Signatures are used to specify the interfaces, or types, of structures, by specifying their individual components. A type component may be specified loosely, permitting a variety of *realisations*, or concretely, by equating it with a particular definition. The latter form supports the specification of *sharing* between type components defined in different structures. A structure *matches* a signature if it provides an implementation for all of the specified components, and possibly more. A signature may be used to *curtail* a matching structure. This restricts access to only those components specified in the signature, while preserving the actual realisations of loosely specified types. A signature may also be used to *abstract* a matching structure, restricting access to components but also *generating* new, and thus abstract, types for loosely specified type components. This provides support for the informal notion of data abstraction.

Finally, Modules allows the definition of parameterised structures called *functors*. A functor is a first-order mapping from structures to structures. A functor is created by specifying a signature for its formal argument, and supplying a structure body defined with respect to this argument. A functor may be *applied* to any structure that matches its argument's signature, resulting in a concrete implementation of the functor body. The actual realisations of the argument's type components are *propagated* to the result. Any abstract types defined in the functor body are generated afresh each time the functor is applied. Functors may be used to express design abstractions and allow flexible code re-use.

The Modules and Core languages are *stratified* in the sense that modules, i.e. functors and structures, may not be manipulated as ordinary values of the Core. This is a limitation, since it means that the architecture of a software system cannot be reconfigured according to run-time demands.

Standard ML is distinguished from most other programming languages by the existence of a formal definition of both its syntax and semantics. It is a strongly typed language with static type checking of programs performed prior to program execution. These two phases are defined, respectively, by separate static and dynamic semantics. Type checking ensures the absence of certain run-time type errors, such as accessing an undefined component of a structure, or using a Core value component at a type incompatible with its definition. In this sense, Standard ML is similar to the formal languages studied in Type Theory. This field of logic has close connections to Computer Science and provides a rational basis for the design of programming languages. Indeed, the type-theoretic underpinnings of the Core language are well-understood.

In recent years, largely inspired by the success of Standard ML Modules, the study of the type-theoretic foundations of module languages has become an active topic of research. Nevertheless, despite numerous attempts, a proper type-theoretic understanding of Standard ML Modules, and its static semantics in particular, has remained elusive. The benefits of a type-theoretic understanding are twofold. Type Theory provides us with a framework for analysing existing features of the language and for synthesising new features by generalisation.

1.1 Objectives and Approach

This thesis has two main objectives. The first is to provide a better, more type-theoretic formulation of the *static semantics* of Modules. The second is to use this formulation as the rational basis for designing proper extensions of the static semantics.

We stress that the primary concern of this thesis is the static semantics of Modules. Our justification for deliberately omitting the dynamic semantics of Modules is that it is already well-understood; moreover, adapting the dynamic semantics to support our static extensions is straightforward.

Our approach is to use concepts from Type Theory as a guideline for reformulating the *existing* semantics of Modules, resulting in a provably equivalent, but hopefully more declarative, accessible and generalisable presentation. The benefit of this approach is that our subsequent extensions to the language can be readily integrated with the existing definition and implementations of Standard ML.

This approach is rather different from those of other researchers in the area. Reductionist approaches aimed at providing type-theoretic semantics of Modules by a translation into existing type theories have either failed to capture significant features and properties of the language, or imposed severe limitations inherited from the chosen model. Others approaches have relied on introducing new, and often badly behaved, type-theoretic constructs.

We therefore see no distinct advantage in abandoning the existing formalism. Instead, we adhere to the existing semantics as much as possible, deviating from the original presentation only when this serves to simplify or clarify it.

1.2 Thesis Outline

In Chapter 2 we give a brief overview of Standard ML, an introduction to relevant concepts from Type Theory, and a survey of related work.

In Chapter 3 we set the scene for the remainder of this thesis by presenting the static semantics of a Modules and Core language in the style of the definition of Standard ML [MTH90, MTH96]. The two languages are presented separately. We first present the Modules language. It models the main features of Standard ML's Modules language. Modules makes relatively few assumptions on the structure of the Core: our definition is parameterised by an arbitrary Core language. For concreteness, we also present a particular instance of the Core language: Core-ML. Although much simpler than Standard ML's Core, it nevertheless captures those features of the language that are relevant to the definition of Modules. Finally, we define Mini-SML as the language obtained by combining the definitions of Modules and Core-ML. We then proceed with an informal analysis of the type-theoretic underpinnings of Mini-SML. In particular, we find no evidence to support the often-made claim that a type-theoretic model of Modules requires *first-order* dependent types. Apart from the feature of *type generativity*, that lends the static semantics a procedural flavour by requiring it to manage a *state* of generated type variables during type checking, the type structure of Modules is easily explained in terms of the simpler, *second-order* notions of type parameterisation, universal quantification over types, and subtyping.

In Chapter 4 we present a new, declarative static semantics for Modules. It is intended as a more type-theoretic alternative to the semantics given in Chapter 3. Our main objective is to first explain and then eliminate the state of generative type variables maintained by the static semantics of Chapter 3. In particular, we reveal that type generativity is no more than a particularly procedural implementation of existential quantification over types. We prove that the two static semantics are equivalent. We claim that the new presentation is more type-theoretic in style, and easier to understand. In subsequent chapters, we substantiate this claim by using this semantics as the basis for significant extensions to the language.

In Chapter 5, we extend the Modules language of Chapter 3 to higher-order. Functors are given the same status currently enjoyed by structures: they may be bound as components of structures, specified as functor arguments and returned as functor results. We give a sound and complete algorithm for signature matching that forms the basis of a type checking algorithm for Higher-Order Modules. This chapter builds on the alternative semantics of generativity given in Chapter 4 and on previous work by Biswas [Bis95] that extends a skeletal fragment of Modules to higher-order. We reformulate, generalise and clarify his definitions, and use them to prove analogous results. Our work furthers his by capturing a notion of type generativity, and by catering for more realistic Core languages, e.g. languages supporting polymorphic values and parameterised types.

In Chapter 6 we briefly discuss the foundations of a separate compilation system for Modules. One of the main criticisms of Standard ML Modules is its perceived lack of support for separate compilation. We review the simple approach to separate compilation in traditional programming languages and explain why previous attempts to adopt this approach in Standard ML have failed. Unlike other researchers, we place the blame for this failure on an inappropriate choice of compilation unit, not on the semantics of Modules.

Instead, we identify an alternative notion of compilation unit that satisfies the requirements of separate compilation. Although acceptable in practice, from a theoretical perspective this solution is only partial. After analysing the problem, we suggest appropriate modifications to the semantics, which are formalised for a skeletal higher-order modules calculus. The adequacy of these modifications is expressed by a theorem, whose proof is sketched.

In Chapter 7 we turn our attention to the particular Core language presented in Chapter 3, Core-ML, and consider relaxing the stratification between Core and Modules. We obtain a language with first-class modules: modules may be passed as arguments to Core-ML functions, returned as results of arbitrary computations, selected using conditional expressions, stored in data structures and so on. Our approach is novel in maintaining the distinction between Core and Modules. Instead of amalgamating the features of both in a *single* language, we provide constructs for packing and unpacking module terms as Core values, allowing programs to alternate between Core and Modules level computation.

In Chapter 8 we consider the type inference problem posed by Core-ML in the presence of both Higher-Order and First-Class Modules. We first review the classical, unification-based type inference algorithm for ML, the language on which Core-ML is based. We then discuss why the naive combination of the type checker for Modules with the traditional type inference algorithm for ML is inadequate. We design a suitably generalised unification algorithm, and present a derived, hybrid typing algorithm that combines type checking of Modules with type inference for Core-ML. We state correctness properties of these algorithms but leave their verification to future work. The algorithms have been tested in a prototype implementation.

Chapter 9 concludes this thesis with a summary of our achievements, a comparison with related work, and directions for future research.

Chapter 10, which was written five years after this thesis was completed, is an epilogue that relates these results to the author's subsequent work on elaborating and implementing the proposed extensions to Modules in Moscow ML [RRS00], a widely used Standard ML compiler.

1.3 Implementation

A prototype interpreter for higher-order and first-class Modules, using Core-ML as a Core language, is available electronically [Rus98a]. It implements the static semantics of Chapters 5 and 7 using a literal implementation of the type inference algorithms in Chapter 8. It also implements a straightforward dynamic semantics, allowing programs to be executed. The Core is enriched with recursion and a smattering of base types to support simple programming examples. In addition, the Modules language supports the definition of signature identifiers, and allows signatures (and functor signatures) to be refined by imposing equational constraints on loosely specified type components. The implementation has been used to check all of the examples in Chapters 5 and 7.

Chapter 2

Background

In this chapter, we give a brief overview of Standard ML (Section 2.1), an introduction to relevant concepts from Type Theory (Section 2.2), and a survey of related work (Section 2.3).

2.1 An Overview of Standard ML

The aim of this section is to briefly review the main features of the programming language Standard ML. A terse and completely formal semantics of Standard ML appears in [MTH90]; [MT91] is an extensive commentary on this definition, sketching some of its meta-theory. A revised definition, with the two-fold aim of simplifying the semantics and meeting some user requests, appears in [MTH96]. Numerous research reports and textbooks provide more tutorial introductions to programming in Standard ML. We recommend the textbook by Paulson [Pau91] and the report by Tofte [Tof89]. Our overview is intentionally informal — the syntax and static semantics of the language will be formalised in Chapter 3. Note that the syntax of our examples, although consistent with the rest of this thesis, deviates slightly from Standard ML. The motivation for this departure is to clearly demarcate the grammar of the Core from the grammar of Modules, facilitating the work in subsequent chapters.

2.1.1 The Standard ML Core language

Standard ML's Core is a strongly typed language. Core terms, or programs, are required to obey the typing rules of the Core before being evaluated (executed). The typing rules are *sound* in the sense that evaluation of any well-typed term is guaranteed to produce no run-time type errors. Rather than present Standard ML's Core, we will sketch a simpler language which shares enough of its features to enable us to present interesting examples of Modules.

Our Core comes with a number of built-in types and terms for creating and manipulating values of those types. For instance, the type `int` supports an infinite set of constants ... `-2`, `-1`, `0`, `1`, `2`... of type `int`. The term `ifzero i e e'` tests whether the term `i`, which must have type `int`, evaluates to zero. If it does, `e` is evaluated, otherwise `e'` is evaluated. The terms `e` and `e'` must have the same type.

The Core is a *functional* language. For example, the built-in function `+` has type `int`$\rightarrow$`(int`$\rightarrow$`int)`. When *applied* to two integer arguments, e.g. `+ 1 2`, it returns their sum, e.g. `3`. We can define our own functions using *parameterised* terms. The term λ`x. + x 1` defines the successor function on integers. The variable `x` is the formal argument or parameter of the function. The type of the successor function is `int`$\rightarrow$`int`, denoting the function space on integers. Even though the Core is strongly typed, we do not have to specify the type of the parameter `x`. Instead, its type is *inferred* from the way in which it is used; in this case, `x` is used as an integer argument of `+`. Core functions can be *higher-order*, in the sense that they may take functions as arguments and return functions as results.

Core terms can be *polymorphic*. For instance, the identity function λ`x.x` can be given the type u$\rightarrow$u, for any type u. The Core uses type *variables* `'a`, `'b`, ... to represent indeterminate types. By universally quantifying over indeterminates in the type of a term, we obtain a schematic description, called a *type scheme*, of all of a term's possible types. For instance, the possible types for the identity are captured by the type scheme $\forall$`'a.'a`$\rightarrow$`'a`, read "for any type `'a`, `'a`$\rightarrow$`'a`". A specific *instance* of a polymorphic type is obtained by substituting actual types for its quantified variables. For example, $\forall$`'a.'a`$\rightarrow$`'a` has instances `int`$\rightarrow$`int`, `bool`$\rightarrow$`bool` and `(int`$\rightarrow$`int)`$\rightarrow$`(int`$\rightarrow$`int)`, obtained by substituting `int`, `bool`, and `(int`$\rightarrow$`int)` for `'a`. A polymorphic term may be used at any type that is an instance of its type scheme. Some type schemes are more general than others. For example, the identity can also be given the type scheme $\forall$`'a 'b.('a`$\rightarrow$`'b)`$\rightarrow$`('a`$\rightarrow$`'b)`. However, $\forall$`'a.'a`$\rightarrow$`'a` is more general than $\forall$`'a 'b.('a`$\rightarrow$`'b)`$\rightarrow$`('a`$\rightarrow$`'b)` because the first scheme has all the instances of the second one, plus some more.

In the Core, we can define *recursive* functions using a fix-point combinator `fix` with type scheme $\forall$`'a 'b.(('a`$\rightarrow$`'b)`$\rightarrow$`('a`$\rightarrow$`'b))`$\rightarrow$`('a`$\rightarrow$`'b)`. Intuitively, `fix f x` evaluates to `f (fix f) x`. For example, consider the higher-order function:

```
λb. λf. fix (λiterbf. λx. ifzero x b (f (iterbf (+ x (- 1)))))
```

When applied to actual arguments `b`, `f` and `x`, it returns the result of the `x`-fold application of the function `f` to the base case `b`. Of course, `x` must be a positive integer, otherwise the function *diverges*. The most general type scheme of this function is:

$$\forall\texttt{'a.'a}\rightarrow\texttt{('a}\rightarrow\texttt{'a)}\rightarrow\texttt{(int}\rightarrow\texttt{'a)}.$$

In Modules, we can *define* a *value identifier* by binding it to the value of a Core term. The identifier can then be used in subsequent phrases. In particular, we can exploit the definition's polymorphism by using the identifier at different types.

The Core supports *parameterised* types. For instance, the type `list` u is the type of lists of elements of type u. We can construct lists using the polymorphic term constants `nil`, of type $\forall$`'a.list 'a`, and `cons`, of type $\forall$`'a.'a`$\rightarrow$`(list 'a)`$\rightarrow$`(list 'a)`. The phrase `list`, on its own, is not a type, but a phrase which, when applied to a type argument, yields a type: `list` is a parameterised type. Since `list` expects a single type argument, we say that its arity or *kind* is `1`.

We can express our own parameterised types by using type phrases of the form Λ`('`$\texttt{a}_1$ $\cdots$ `'`$\texttt{a}_\text{k}$`)`.u. The prefix Λ declares the variables `'`$\texttt{a}_1$ through `'`$\texttt{a}_\text{k}$ as parameters of the type u. We say that the parameterised type has arity or *kind* k, since it expects k type arguments. For instance, we can use the parameterised type Λ`('a,'b,'c).('b`$\rightarrow$`'c)`$\rightarrow$`('a`$\rightarrow$`'b)`$\rightarrow$`('a`$\rightarrow$`'c)` as a uniform description of a family of types: when applied to any three type arguments, it returns the type of a function that takes two functions and returns another.

Given that Core terms are implicitly typed, the ability to express parameterised types does not seem very useful. However, Modules is *explicitly* typed and requires some syntax in order to specify Core type and value components. Fortunately, in Modules we can *define* a *type identifier* by binding it to a parameterised type. The identifier can then be used as a proper abbreviation, either in the definition of another type identifier, or in the specification of a value component's type.

Standard ML's Core has a number of other features not illustrated here. Indeed, it comes with a rich collection of basic types, supports the definition of mutually recursive types and functions, has first-class, dynamically allocated references (typed pointers), exceptions and pattern-matching. Most

```
     type nat = int;
     val zero = 0;
     val succ = λx.+ x 1;
     val iter = λb.λf.
         fix (λiterbf.λx.ifzero x b (f (iterbf (x + (- 1)))));
     ...
     val even = λn.iter true not n
```

Figure 2.1: A sequence of Core type and term definitions.

```
structure IntNat =
 struct type nat = int;
        val zero = 0;
        val succ = λx.+ x 1;
        val iter = λb.λf.
          fix (λiterbf.λx.ifzero x b (f (iterbf (+ x (- 1)))))
 end
```

Figure 2.2: An implementation of the natural numbers represented as integers.

of these features have little or no interaction with Modules, and we shall not consider them any further in this thesis.

2.1.2 The Standard ML Modules Language

At the most basic level, Modules extends the Core language with a facility for *defining type identifiers* denoting parameterised types, and *value identifiers* denoting the values of Core terms. Subsequent definitions may refer to previously defined identifiers. The reason we view definitions as part of Modules and not the Core is that they also give rise to module components, as we shall see shortly.

For instance, the sequence of definitions in Figure 2.1 defines the type identifier `nat` (with no parameters), and the value identifiers `zero`, `succ` (the successor function) and `iter` (an iterator). The definition of `even` uses `iter` to give a simple-minded test for whether a (positive) integer is even.

Writing a large program as a long sequence of mostly unrelated definitions quickly becomes unmanageable. In Modules, we can encapsulate a

```
structure IntNatAdd =
        struct structure Nat = IntNat;
               val add = λn.λm.(Nat.iter) n (Nat.succ) m
        end
```

Figure 2.3: A structure with a substructure.

```
   sig type nat = int;
       val zero : nat;
       val succ : nat→nat;
       val iter : ∀'a.'a → ('a → 'a) → (nat →'a)
   end
```

Figure 2.4: The signature of a structure implementing naturals using the integers.

body of Core definitions into a unit called a *structure*, by surrounding it with the keywords `struct` and `end`. We can then bind this anonymous structure to a *structure identifier*. In Figure 2.2, the identifier `IntNat` refers to the collection of components in its definition. Intuitively, `IntNat` defines an implementation of the natural numbers as a subset of the integers.

The components of a structure identifier can be accessed by using the *dot notation*. For instance, the type `IntNat.nat` refers to the type component `nat` of `IntNat` and *denotes* the type `int`. The term `IntNat.zero` refers to the value component `zero` of `IntNat`. It evaluates to `0`, and has type `int`.

Structure bodies may themselves contain definitions of (sub)structures. We can use this to express the architecture of a system as a *hierarchy* of components. Figure 2.3 defines a structure `IntNatAdd` with one substructure `Nat` and an addition function derived from `Nat`. Component structures are also accessed via the dot notation, e.g. `IntNatAdd.Nat`. By iterating the dot notation, we can refer to type, term and structure components defined at arbitrary depths. Component identifiers must be uniquely defined within a structure body, but may be reused within different substructures, providing a good mechanism for name space control.

Signature expressions are used to specify the types of structures, by listing the specifications of their components. A signature expression consists of a *body* of component specifications, encapsulated by the key words `sig` and

```
   sig type nat : 0;
       val zero : nat;
       val succ : nat → nat;
       val iter : ∀'a.'a → ('a → 'a) → (nat →'a)
   end
```

Figure 2.5: The signature of a structure implementing naturals using an arbitrary type.

```
structure ResIntNat =
   IntNat ⪰ sig type nat : 0;
                 val succ : nat → nat;
                 val iter : nat → (nat → nat) → (nat → nat)
             end
```

Figure 2.6: A curtailed view of `IntNat`.

`end`. Subsequent specifications may refer to previously defined and specified identifiers. The signature in Figure 2.4 specifies the type of a structure implementing naturals using the given definition `int` for the type component `nat`. The signature in Figure 2.5, on the other hand, specifies the type of a structure implementing naturals using some definition of kind `0` for the type component `nat`. Observe that a type component may be specified in one of two ways, either by specifying its actual definition, or, more flexibly, by only specifying its kind, permitting a variety of actual definitions.

Roughly speaking, a structure expression *matches* a signature if it implements all of the components specified in the signature. In particular, the structure must *realise* all of the type components that are merely specified but not defined in the signature. Moreover, the structure must *enrich* the signature subject to this realisation: every specified type must be implemented by an equivalent type; every specified value must be implemented by a value whose type is at least as general as its specification; finally, every specified structure must be implemented by a structure that enriches its specification. The order in which components of the structure are actually defined is irrelevant. Furthermore, the structure is free to define more components than specified in the signature.

Signatures play a number of different roles.

```
structure AbsNat =
    IntNat \ sig type nat : 0;
                 val zero : nat;
                 val succ : nat → nat;
                 val iter : ∀'a.'a → ('a → 'a) → (nat → 'a)
             end
```

Figure 2.7: An abstract view of `IntNat`.

In Figure 2.6 we define a new structure `ResIntNat` corresponding to a restricted view of `IntNat`. The infix operator $\succeq$ means that the signature, which is matched by `IntNat`, is used to *curtail* its implementation by hiding its `zero` component and restricting the polymorphism of its `iter` component. However, the actual realisation of the type component `ResIntNat.nat` by the type `int` remains *transparent*, even though its definition is not specified in the signature. For instance the application `ResIntNat.succ (-3)` is still well-typed, because `-3` has type `int`. Note, however, that `-3` does not correspond to the representation of a natural number.

In Figure 2.7 we define a new structure `AbsNat` corresponding to an abstract view of `IntNat`. The infix operator $\backslash$ means that the signature, which is matched by `IntNat`, is used to *abstract* its implementation by generating a *new* type for the specified type `nat`. The realisation of the type component `nat` by the type `int` is effectively forgotten. In this way, `AbsNat` defines an *abstract datatype* of natural numbers. For instance the application `AbsNat.succ (-3)` is no longer well-typed, since `-3` has type `int` but `AbsNat.succ` expects a value of the abstract type `AbsNat.nat`, e.g. `AbsNat.zero`. In general, abstractions also have a curtailment effect, although this is not illustrated in this example.

Finally, Standard ML supports the definition of parameterised structures called *functors*. Intuitively, a functor is a function mapping structures to structures. Figure 2.8 defines a functor called `AddFun` with formal parameter `N`, which is assumed to match the specified signature. The structure expression to the right of the `=` sign is the *body* of the functor and refers to the formal argument. The body may assume no more information about `N` than is specified in its signature. This provides another form of abstraction: the functor `AddFun` can be defined before any structure implementing the naturals has been written.

A functor is used to create a structure by *applying* it to an actual ar-

```
functor AddFun(N:sig type nat : 0;
                     val zero : nat;
                     val succ : nat → nat;
                     val iter : ∀'a.'a → ('a → 'a) →
                                     (nat → 'a)
                 end) =
        struct structure Nat = N;
               val add = λn.λm. (Nat.iter) n (Nat.succ) m
        end
in

...

structure IntNatAdd = AddFun IntNat;
structure AbsNatAdd = AddFun AbsNat
```

Figure 2.8: A functor and its application.

gument. The actual argument must match the formal argument's signature. Figure 2.8 shows two different applications of `AddFun`. The definition of `IntNatAdd` evaluates to the same implementation as `IntNatAdd` in Figure 2.3. Observe that `AddFun` is applied twice, to arguments that actually differ in the implementation of their type component `nat` (recall that `AbsNat.nat` is an abstract type distinct from `int`). Moreover, each application *propagates* the actual realisation of the specified type component. Thus we can exploit the fact that `IntNatAdd.Nat.nat` is actually `int`, and that `AbsNatAdd.Nat.nat` is the same as the abstract type `AbsNat.nat`.

The functor `GenFun` in Figure 2.9 illustrates a different property of functors. `GenFun` almost defines an identity function on natural number structures, except that it returns the result of *abstracting* its argument by its signature. In particular, each application `GenFun` generates a new abstract type: we call this the *generative* property of functor application. For example, the types `X.nat` and `Y.nat` are incompatible, even though `GenFun` is applied to the *same* argument in each case.

In general, functors allow us to decompose a large programming task into separate subtasks which may be implemented in isolation. The propagation of type realisations means that we can use functors to extend existing types with operations compatible with those types. The generative property of

```
functor GenFun(N:sig type nat : 0;
                     val zero : nat;
                     val succ : nat → nat;
                     val iter : ∀'a.'a → ('a → 'a) →
                                    (nat → 'a)
                 end) =
         N \ sig type nat : 0;
                 val zero : nat;
                 val succ : nat → nat;
                 val iter : ∀'a.'a → ('a → 'a) → (nat → 'a)
             end
in
structure X = GenFun IntNat;
structure Y = GenFun IntNat
```

Figure 2.9: A generative functor.

functors ensures that conceptually distinct applications of the same functor return distinct abstract types. In Standard ML, a functor may only be defined at the outermost or top-level of a program. In particular, a functor may not be defined as a component of a structure, applied to a functor, or return another functor as a result. These restrictions mean that functors are *first-order* mappings on structures.

Functors are commonly used to combine structures. Frequently, these structures need to interact via values of a *shared* type. Consider the example in Figure 2.10. The functor `SQ` is a first attempt to implement a sum-of-squares function using the addition and multiplication functions provided by its argument's substructures, `AddNat` and `MultNat`. Unfortunately, it fails to type-check because the argument's signature does not specify that these substructures must share the same representation of natural numbers. In the original version of Standard ML [MTH90], we could specify the required sharing by inserting a *type sharing constraint* `sharing type AddNat.Nat.nat = MultNat.Nat.nat` at the end of the argument's signature. Another way of achieving this, adopted in the revision of Standard ML [MTH96], and illustrated in Figure 2.11, is to specify the sharing directly at the specification of `MultNat.Nat.nat` by using the concrete specification `type nat = AddNat.Nat.nat`.

Remark 2.1.1. Originally, Standard ML [MTH90] also supported an addi-

```
functor SQ(X:sig structure AddNat :
                   sig structure Nat :
                         sig type nat : 0
                         end;
                       val add: Nat.nat → Nat.nat → Nat.nat
                   end;
                 structure MultNat :
                   sig structure Nat :
                         sig type nat : 0
                         end;
                       val mult: Nat.nat → Nat.nat → Nat.nat
                   end
             end) =
 struct val sumsquare =
             λn.λm. X.AddNat.add (X.MultNat.mult n n)
                                 (X.MultNat.mult m m)
 end;
```

Figure 2.10: A functor that fails to type-check due to inadequate sharing.

```
functor SQ(X:sig structure AddNat :
                   sig structure Nat :
                         sig type nat : 0
                         end;
                       val add: Nat.nat → Nat.nat → Nat.nat
                   end;
                 structure MultNat :
                   sig structure Nat :
                         sig  type nat = AddNat.Nat.nat
                         end;
                       val mult: Nat.nat → Nat.nat → Nat.nat
                   end
             end) =
 struct val sumsquare =
             λn.λm. X.AddNat.add (X.MultNat.mult n n)
                                 (X.MultNat.mult m m)
 end;
```

Figure 2.11: A functor that type-checks thanks to specified sharing.

tional, stronger notion of sharing, called *structure sharing*. Encapsulating a body of declarations within `struct` and `end` generated a new internal *structure name*, or stamp, for the expression, similar to the generation of a fresh abstract type. However, the name identified the entire structure, not just individual types. In signatures, *structure sharing constraints* could be used to specify structure components with shared names. This made it possible to specify sharing not only of types, but also of values, since any two structures which shared the same name must have originated from a common ancestor. This interesting but little used feature has been abandoned in the revised definition of Standard ML [MTH96]. The move simplifies the semantics, benefitting both the working programmer and language implementor.

2.1.3 Summary

At first glance, the Modules language appears to be nothing more than a small, explicitly typed functional language. On closer inspection, however, this analogy breaks down because Modules has many interesting features that are not accounted for in conventional functional programming languages. The ability to define structures, containing definitions of both types and values, is novel, and raises the question of what it means to project a type component from a structure: in particular, do we need to evaluate the structure at *run-time* to determine the meaning of the type component, or is it sufficient to know the *compile-time* type of the structure. The syntax of Modules seems to suggest that signatures are the types of structures, but this cannot really be the case, because signatures can contain loosely specified type components. For instance, the signature of a functor's formal argument may not fully determine the functor's domain, while the declared type of a function's formal argument typically does determine the function's domain. Indeed, a functor can be applied to any argument that both realises and enriches its argument signature. This is much more flexible than the usual notion of function application, that requires the domain of the function and the type of its actual argument to be equal. Finally, the fact that curtailing a structure by a signature does not hide the actual realisation of the signature's type components means that the curtailment phrase cannot simply be regarded as the analog of a type constraint in a functional language. The analogy does not hold for the abstraction phrase either, because it can generate new types.

2.2 An Introduction to Type Theory

Although the published definition of Standard ML [MTH90, MT91, MTH96] formally defines the Modules language, it makes few concessions to help the reader understand its features. It particular, no attempt is made to relate these features to well-known concepts developed in the theory of programming languages. This has presented an obstacle not only to the understanding of the language, but also to its further development.

Type Theory provides a framework for discussing issues of programming language semantics. Mitchell's textbook [Mit96] is a good introduction to the use of Type Theory as a foundation for programming language analysis and design. In this thesis, we use Type Theory as a guideline to both clarify and generalise the static semantics of Modules. Our first step will be to recast the static semantics of Modules using well-known concepts from Type Theory. The purpose of this section is to provide a gentle introduction to these concepts. We shall illustrate the ideas with examples that deliberately evoke the examples used to present Standard ML in Section 2.1.

In its most general sense, a *type theory* is a formal language based on a conceptual organisation of phrases in the language. A *type* is a phrase that describes a collection of phrases with a common property: the type of natural numbers, the type of pairs, the type of functions, and so on. It is typically possible to distinguish between, on the one hand, the syntactic class of *term phrases* exhibiting properties, and, on the other hand, the syntactic class of *type phrases* describing these properties. The basic statements, or *judgements*, we make in type theory are concerned with the *classification* of term phrases (e.g. does a term have a type) and the *equivalence* of two well-typed term phrases (e.g. are two terms equivalent at a type). In more complicated type theories, we may even distinguish between different *kinds* of type phrases, and make similar statements regarding the classification and equivalence of type phrases. Typically, each judgement is captured by a *relation*, whose definition is specified by a set of *inference rules*.

Type Theory has its origins in logic. Indeed, many type theories exhibit a close correspondence with particular systems of constructive logic. For such theories, types correspond to propositions of some logic and the rules defining the well-typedness of terms can be put in one-to-one relation with the inference rules of the logic. Such a correspondence is known as a *Curry-Howard isomorphism*, and gives rise to the interpretation of "propositions as types" [CF58, How80]. Each proposition is identified with a type, and each proof of the proposition with a term of that type. The correspondence goes deeper than this because the notions of equivalent proofs and equivalent terms also coincide. Thompson's textbook [Tho91] gives a nice introduction to the logical interpretation of Type Theory.

There is another interpretation of Type Theory that is more closely related to Computer Science. In this interpretation, terms are *programs*, and types are their *specifications*. The equational judgements between terms capture program equivalences that may be directed to yield a notion of program execution. In the special case where the type theory corresponds to a constructive logic, we can think of the terms of the theory as programs for constructing canonical proofs of their types.

For typed, general-purpose programming languages, that is languages with state, non-termination, exceptions and other features, the connections with logic are more tenuous. Nevertheless, the notion of type remains useful as it can be viewed as a *partial* specification of a program's behaviour. Typically, the well-typedness of a program will guarantee the absence of certain run-time errors corresponding to type violations such as using a natural number as a function, or accessing an undefined field of a record. In an *untyped* programming language, these run-time errors are either ignored at one's peril, or must be detected and trapped at run-time, incurring an additional overhead. Still, a program's type may tell us nothing at all about the program's other properties such as termination, effect on the store or exceptional behaviour. It is in this sense that types are merely partial specifications.

The following sections give a whistle-stop tour of some basic type theoretic constructs. We will focus on the kinds of programs these theories allow us to express. The notation we use is inspired by the logical origins of each construct, to which we shall allude whenever appropriate. We will start with the simply typed λ-calculus, and then consider a sequence of orthogonal but compatible extensions to it.

2.2.1 The Simply Typed Lambda Calculus

Figure 2.12 defines the phrase classes and grammar of the simply typed λ-calculus (with type variables). *Kind* phrases $\kappa \in \mathrm{Kind}$ are used to classify *type* phrases $\tau \in \mathrm{Type}$. The meta-variable α ranges over an infinite set of *type variables*. Type phrases of kind $\star$ (read "type") are used to classify *term* phrases $\mathrm{e} \in \mathrm{Term}$. The set of terms defines a language of explicitly typed functions. The meta-variable x ranges over an infinite set of *term variables*. A context $\mathrm{C} \in \mathrm{Context}$ is a finite sequence of declarations (or assumptions) relating term variables to their types and type variables to their kinds. Since there is only a single kind $\star \in \mathrm{Kind}$, the set of kinds is superfluous in this

$\kappa \in$ Kind	::=	$\star$	types classifying terms
$\tau \in$ Type	::=	α	type variable
	$\mid$	$\tau \rightarrow \tau'$	function space
e $\in$ Term	::=	x	term variable
	$\mid$	λx:τ.e	function
	$\mid$	e e$'$	application
C $\in$ Context	::=	$\epsilon_{\mathcal{C}}$	empty context
	$\mid$	C,x:τ	term declaration
	$\mid$	C,α:κ	type declaration

Figure 2.12: Grammar of the simply typed λ-calculus.

theory; it is included here because we shall generalise it in later sections.

The judgements defining the theory of the simply typed λ-calculus are defined as the least relations closed under the inference rules in Figure 2.13. Because type and term phrases can contain free variables, our judgements are relative to a context of assumptions concerning the classifications of free variables.

The judgement $\vdash$ C **valid** defines the set of *valid* contexts. The notation Dom(C) is used to denote the set of variables declared in the context C. A context is valid provided each variable in its domain is uniquely declared, and the classification of each variable is well-formed with respect to the preceding declarations in the context. The other judgments are formulated in a way that ensures that contexts are valid. Note that a term variable's type must have kind $\star$.

The judgement C $\vdash \kappa$ **kind**, read "in context C, κ is a kind", defines the set of *valid* kinds. In this system, the judgement happens to be trivial because we only have the single kind $\star$.

The judgement C $\vdash \tau : \kappa$, read "in context C, type τ has kind κ", classifies a type by its kind. A type variable α has a kind provided it is declared with that kind in the context. The *function space* $\tau \rightarrow \tau'$, classifying the collection of functions with *domain* τ and *range* τ', has kind $\star$ provided the domain and range have kind $\star$.

The judgement C $\vdash$ e : κ, read "in context C, term e has type τ",

Valid Contexts $\boxed{\vdash \mathrm{C}\ \textbf{valid}}$

$$\frac{}{\vdash \epsilon_{\mathcal{C}}\ \textbf{valid}}$$

$$\frac{\mathrm{C} \vdash \kappa\ \textbf{kind} \quad \alpha \notin \mathrm{Dom}(\mathrm{C})}{\vdash \mathrm{C}, \alpha : \kappa\ \textbf{valid}}$$

$$\frac{\mathrm{C} \vdash \tau : \star \quad \mathrm{x} \notin \mathrm{Dom}(\mathrm{C})}{\vdash \mathrm{C}, \mathrm{x} : \tau\ \textbf{valid}}$$

Valid Kinds $\boxed{\mathrm{C} \vdash \kappa\ \textbf{kind}}$

$$\frac{\vdash \mathrm{C}\ \textbf{valid}}{\mathrm{C} \vdash \star\ \textbf{kind}}$$

Type Classification $\boxed{\mathrm{C} \vdash \tau : \kappa}$

$$\frac{\vdash \mathrm{C}, \alpha : \kappa, \mathrm{C}'\ \textbf{valid}}{\mathrm{C}, \alpha : \kappa, \mathrm{C}' \vdash \alpha : \kappa}$$

$$\frac{\mathrm{C} \vdash \tau : \star \quad \mathrm{C} \vdash \tau' : \star}{\mathrm{C} \vdash \tau \to \tau' : \star}$$

Term Classification $\boxed{\mathrm{C} \vdash \mathrm{e} : \tau}$

$$\frac{\vdash \mathrm{C}, \mathrm{x} : \tau, \mathrm{C}'\ \textbf{valid}}{\mathrm{C}, \mathrm{x} : \tau, \mathrm{C}' \vdash \mathrm{x} : \tau}$$

$$\frac{\mathrm{C}, \mathrm{x} : \tau \vdash \mathrm{e} : \tau'}{\mathrm{C} \vdash \lambda \mathrm{x}{:}\tau.\mathrm{e} : \tau \to \tau'}$$

$$\frac{\mathrm{C} \vdash \mathrm{e} : \tau' \to \tau \quad \mathrm{C} \vdash \mathrm{e}' : \tau'}{\mathrm{C} \vdash \mathrm{e}\ \mathrm{e}' : \tau}$$

Term Equivalence $\boxed{\mathrm{C} \vdash \mathrm{e} = \mathrm{e}' : \tau}$

$$\frac{\mathrm{C} \vdash \lambda \mathrm{x}{:}\tau'.\mathrm{e} : \tau' \to \tau \quad \mathrm{C} \vdash \mathrm{e}' : \tau'}{\mathrm{C} \vdash (\lambda \mathrm{x}{:}\tau'.\mathrm{e})\ \mathrm{e}' = [\mathrm{e}'/\mathrm{x}]\,(\mathrm{e}) : \tau} \quad (\beta)$$

(rules for congruence, symmetry, reflexivity and transitivity omitted)

Figure 2.13: Judgements of the simply typed λ-calculus.

classifies a term by its type. A term variable x has a type provided it is declared with that type in the context. A *function* $\lambda \mathrm{x}{:}\tau.\mathrm{e}$ has type $\tau \to \tau'$ provided its body e has the type τ' under the additional assumption that the *parameter* x has the declared type τ. An *application* e e$'$ has type τ, provided e is a function with domain τ' and range τ, and e$'$ has type τ'.

The final judgement $\mathrm{C} \vdash \mathrm{e} = \mathrm{e}' : \tau$, read "in context C, term e is equivalent to term e$'$ at type τ" defines the notion of equivalence between terms of the same type. We have only presented the key rule, Rule (β), that equates a function application with the term obtained by *substituting* the actual argument for the formal parameter of the function. By ordering this equation from left-to-right we obtain a notion of typed reduction, which can be used to evaluate terms to their normal form (the intuitive notion of a term's *value*). The resulting reduction relation is typed because it still mentions premises requiring the well-typedness of terms. By erasing these premises, we obtain a much more efficient notion of *untyped* reduction. It is a meta-theorem of the theory that untyped reduction from well-typed terms respects the equational judgements.

Example 2.2.1 *(Typical Examples of Well-typed terms).* The simplest example of a function is the identity on terms of type α:

$$\alpha : \star \vdash \lambda \mathrm{x}{:}\alpha.\mathrm{x} : \alpha \to \alpha.$$

A more involved example is the higher-order function that composes two functions f and g:

$$\begin{array}{rcl} \alpha : \star, \beta : \star, \gamma : \star & \vdash & \lambda \mathrm{f}{:}\beta \to \gamma.\lambda \mathrm{g}{:}\alpha \to \beta.\lambda \mathrm{x}{:}\alpha.\mathrm{f}\ (\mathrm{g}\ \mathrm{x}) \\ & : & (\beta \to \gamma) \to (\alpha \to \beta) \to (\alpha \to \gamma) \end{array}$$

Remark 2.2.1 *(The Logical Interpretation).* By ignoring term phrases, reading type variables as atomic *propositions*, and the function space as logical *implication* ($\_ \supset \_$), it is easy to see that the term classification rules correspond to the inference rules of minimal intuitionistic propositional logic.

Remark 2.2.2 *(The Phase Distinction).* A typed programming language is said to obey a *phase distinction* [Car88b] if the type of any term in the language can be checked without evaluating arbitrary terms. This allows the semantics of the language to be split into a *static* semantics of type checking, that is performed at compile-time, and a *dynamic* semantics of evaluation, that is performed at run-time. The phase distinction is important because it ensures that the tractability of type checking is independent of term evaluation, which, in typical programming languages, may fail to terminate. Viewing a type theory as a programming language, we can regard its classification judgements as defining the *type-checking* phase, and its term equivalence judgement as defining the *evaluation* phase. It is easy to see that the simply typed λ-calculus obeys a natural phase distinction because the classification rules for both terms and types are defined *independently* of the term equivalence judgement.

As a programming language, the simply typed λ-calculus is not very interesting. However, it is very easy to extend the calculus by adding new phrases and inference rules. For instance, to define the type of integers we can add the type constant **int**, numeric constants $\bar{i}$ for each integer $i \in \{\ldots, -2, -1, 0, 1, 2, \ldots\}$, addition $+$, negation $-$, and a family of zero tests $\mathbf{ifzero}_\tau$, together with the classification rules:

$$\frac{\vdash \mathrm{C}\ \mathbf{valid}}{\mathrm{C} \vdash \mathbf{int} : \star}$$

$$\frac{\vdash \mathrm{C}\ \mathbf{valid} \quad i \in \{\ldots, -2, -1, 0, 1, 2, \ldots\}}{\mathrm{C} \vdash \bar{i} : \mathbf{int}}$$

$$\frac{\vdash \mathrm{C}\ \mathbf{valid}}{\mathrm{C} \vdash - : \mathbf{int} \to \mathbf{int}}$$

$$\frac{\vdash \mathrm{C}\ \mathbf{valid}}{\mathrm{C} \vdash + : \mathbf{int} \to \mathbf{int} \to \mathbf{int}}$$

$$\frac{\mathrm{C} \vdash \tau : \star}{\mathrm{C} \vdash \mathbf{ifzero}_\tau : \mathbf{int} \to \tau \to \tau \to \tau}$$

and the equational rules:

$$\frac{\mathrm{C} \vdash +\, \bar{i}\ \bar{j} : \mathbf{int}}{\mathrm{C} \vdash +\, \bar{i}\ \bar{j} = \overline{i+j} : \mathbf{int}}$$

$$\frac{\mathrm{C} \vdash -\, \bar{i} : \mathbf{int}}{\mathrm{C} \vdash -\, \bar{i} = \overline{-i} : \mathbf{int}}$$

$$\frac{\mathrm{C} \vdash \mathbf{ifzero}_\tau\ \bar{0}\ \mathrm{e}\ \mathrm{e}' : \tau}{\mathrm{C} \vdash \mathbf{ifzero}_\tau\ \bar{0}\ \mathrm{e}\ \mathrm{e}' = \mathrm{e} : \tau}$$

$$\frac{\mathrm{C} \vdash \mathbf{ifzero}_\tau\ \bar{i}\ \mathrm{e}\ \mathrm{e}' : \tau \quad \bar{i} \not\equiv \bar{0}}{\mathrm{C} \vdash \mathbf{ifzero}_\tau\ \bar{i}\ \mathrm{e}\ \mathrm{e}' = \mathrm{e}' : \tau}$$

where $\bar{i} \equiv \bar{j}$ if, and only if, the numeric constants $\bar{i}$ and $\bar{j}$ are syntactically equal.

It is also possible to axiomatise *structured* types such as pairs, variants, and lists. For instance, we can extend the calculus with *labeled records* of terms by assuming some infinite set of labels $l \in$ Label, and adding record type phrases $\{l_0 : \tau_0, \ldots, l_{k-1} : \tau_{k-1}\}$ (for $k \geq 0$). The term phrase $\{l_0 = \mathrm{e}_0, \ldots, l_{k-1} = \mathrm{e}_{k-1}\}$ (for $k \geq 0$) introduces a record type. The term phrase $\mathrm{e}.l$ eliminates a record type by projecting on a field. We implicitly identify record types that differ only in the ordering of fields. The rules will ensure that all the fields of a record have distinct labels. Let $[k]$ denote the set of indices $\{i \mid 0 \geq i < k\}$ for $k \geq 0$, and let $l \equiv l'$ hold if, and only if, the labels l and l' are syntactically equal. We extend our judgements with the following rules:

$$\frac{\forall i \in [k].\mathrm{C} \vdash \tau_i : \star \quad \forall j \in [i-1].l_i \not\equiv l_j}{\mathrm{C} \vdash \{l_0 : \tau_0, \ldots, l_{k-1} : \tau_{k-1}\} : \star}$$

$$\frac{\forall i \in [k].\mathrm{C} \vdash \mathrm{e}_i : \tau_i \quad \forall j \in [i-1].l_i \not\equiv l_j}{\mathrm{C} \vdash \{l_0 = \mathrm{e}_0, \ldots, l_{k-1} = \mathrm{e}_{k-1}\} : \{l_0 : \tau_0, \ldots, l_{k-1} : \tau_{k-1}\}}$$

$$\frac{\mathrm{C} \vdash \mathrm{e} : \{l_0 : \tau_0, \ldots, l_i : \tau_i, \ldots, l_{k-1} : \tau_{k-1}\} \quad i \in [k]}{\mathrm{C} \vdash \mathrm{e}.l_i : \tau_i}$$

For brevity, we omit the equational rules for records.

More importantly, it is easy to extend the computational power of the calculus. For instance, by introducing fix-point operators, we obtain a language with unbounded recursion. We simply add the term constant $\mathbf{fix}_\tau$, one for each type τ, and the rule schemes:

$$\frac{\mathrm{C} \vdash \tau : \star}{\mathrm{C} \vdash \mathbf{fix}_\tau : (\tau \to \tau) \to \tau}$$

$$\frac{\mathrm{C} \vdash \mathbf{fix}_\tau\ \mathrm{e} : \tau}{\mathrm{C} \vdash \mathbf{fix}_\tau\ \mathrm{e} = \mathrm{e}\ (\mathbf{fix}_\tau\ \mathrm{e}) : \tau}$$

This turns the calculus into a *general-purpose* programming language. Of course, adding fix-points makes the term equivalence judgement undecidable. However, because of the phase distinction, the decidability of the classification judgements is preserved.

Example 2.2.2 *(Programming Examples).* Using λ-abstraction, we can define the successor function on integers as:

$$succ \equiv \lambda x{:}\mathbf{int}.+\ x\ \bar{1}.$$

It is easy to derive:

$$\vdash succ : \mathbf{int} \to \mathbf{int}.$$

We can group zero with the successor function to form a record whose components may be used to generate the natural numbers:

$$\vdash \{\mathbf{z} = \bar{0}, \mathbf{s} = \mathit{succ}\} : \{\mathbf{z} : \mathbf{int}, \mathbf{s} : \mathbf{int} \to \mathbf{int}\}.$$

For a given type τ, we can define a function for iterating functions over τ as follows:

$$\begin{array}{l} \vdash \lambda \mathrm{b}{:}\tau.\lambda \mathrm{f}{:}\tau \to \tau.\mathbf{fix}_{\mathbf{int}\to\tau}\ (\lambda \mathrm{I}{:}\mathbf{int} \to \tau.\lambda \mathrm{i}{:}\mathbf{int}.\mathbf{ifzero}_{\tau}\ \mathrm{i}\ \mathrm{b}\ (\mathrm{f}\ (\mathrm{I}\ (+\ \mathrm{i}\ (-\ \bar{1}))))) \\ :\ \tau \to (\tau \to \tau) \to (\mathbf{int} \to \tau). \end{array}$$

Unfortunately, we have to redefine this function for each choice of τ. This is because we need to appeal to *different* incarnations of the fix point operator and zero test whenever we need a different result type τ.

2.2.2 Quantification over Types of a Kind

From a logical perspective, a natural generalisation of propositional logic is to allow universal and existential *quantification* over propositional variables, leading to second-order propositional logic. Figure 2.14 summarises the additional phrases and rules.

Let us first consider the addition of universally quantified types $\forall\alpha{:}\kappa.\tau$. In the simply-typed λ-calculus, we could parameterise a term by a term, and apply a parameterised term to a term argument. In this extension, we can parameterise a term by a *type*, and apply a type-parametric term to a *type argument*. If e has type τ, then the parameterised term $\Lambda\alpha{:}\kappa.\mathrm{e}$ has type $\forall\alpha{:}\kappa.\tau$, introducing a universal quantifier. If e has type $\forall\alpha{:}\kappa.\tau'$ then the application e $[\tau]$ has type $[\tau/\alpha]\,(\tau')$, eliminating a universal quantifier. We adopt the standard notational convention of using Λ to bind type parameters, to distinguish it from λ that binds term parameters; we also enclose type arguments in square brackets ([_]) to distinguish them from term arguments.

Example 2.2.3 *(Typical Examples of Well-typed terms).* A simple example is the function that, for any type argument, returns the identity function on that type:

$$\vdash \Lambda\alpha{:}\star.\lambda \mathrm{x}{:}\alpha.\mathrm{x} : \forall\alpha{:}\star.\alpha \to \alpha$$

By applying this function, call it *id*, to different type arguments, we obtain different *instances* of the identity function. Thus, assuming **int** and **bool** are types we have:

$$\vdash \mathit{id}\ [\mathbf{int}] : \mathbf{int} \to \mathbf{int},$$

$\tau \in \mathrm{Type}$	$::=$	$\ldots$	
	$\mid$	$\forall\alpha{:}\kappa.\tau$	universal quantification
	$\mid$	$\exists\alpha{:}\kappa.\tau$	existential quantification
$\mathrm{e} \in \mathrm{Term}$	$::=$	$\ldots$	
	$\mid$	$\Lambda\alpha{:}\kappa.\mathrm{e}$	type parameterisation
	$\mid$	$\mathrm{e}\,[\tau]$	type application
	$\mid$	$\mathbf{pack}\ \tau\ \mathrm{e}\ \mathbf{as}\ \exists\alpha{:}\kappa.\tau'$	type abstraction
	$\mid$	$\mathbf{open}\ \mathrm{e}\ \mathbf{as}\ \alpha{:}\kappa, \mathrm{x}{:}\tau\ \mathbf{in}\ \mathrm{e}'$	accessing an abstraction

Type Classification $\boxed{\mathrm{C} \vdash \tau : \kappa}$

$$\frac{\mathrm{C}, \alpha : \kappa \vdash \tau : \star}{\mathrm{C} \vdash \forall\alpha{:}\kappa.\tau : \star} \qquad \frac{\mathrm{C}, \alpha : \kappa \vdash \tau : \star}{\mathrm{C} \vdash \exists\alpha{:}\kappa.\tau : \star}$$

Term Classification $\boxed{\mathrm{C} \vdash \mathrm{e} : \tau}$

$$\frac{\mathrm{C}, \alpha : \kappa \vdash \mathrm{e} : \tau}{\mathrm{C} \vdash \Lambda\alpha{:}\kappa.\mathrm{e} : \forall\alpha{:}\kappa.\tau}$$

$$\frac{\mathrm{C} \vdash \mathrm{e} : \forall\alpha{:}\kappa.\tau' \quad \mathrm{C} \vdash \tau : \kappa}{\mathrm{C} \vdash \mathrm{e}\,[\tau] : [\tau/\alpha]\,(\tau')}$$

$$\frac{\mathrm{C} \vdash \exists\alpha{:}\kappa.\tau' : \star \quad \mathrm{C} \vdash \tau : \kappa \quad \mathrm{C} \vdash \mathrm{e} : [\tau/\alpha]\,(\tau')}{\mathrm{C} \vdash \mathbf{pack}\ \tau\ \mathrm{e}\ \mathbf{as}\ \exists\alpha{:}\kappa.\tau' : \exists\alpha{:}\kappa.\tau'}$$

$$\frac{\mathrm{C} \vdash \exists\alpha{:}\kappa.\tau : \star \quad \mathrm{C} \vdash \mathrm{e} : \exists\alpha{:}\kappa.\tau \quad \mathrm{C}, \alpha : \kappa, \mathrm{x} : \tau \vdash \mathrm{e}' : \tau' \quad \mathrm{C} \vdash \tau' : \star}{\mathrm{C} \vdash \mathbf{open}\ \mathrm{e}\ \mathbf{as}\ \alpha{:}\kappa, \mathrm{x}{:}\tau\ \mathbf{in}\ \mathrm{e}' : \tau'}$$

Term Equivalence $\boxed{\mathrm{C} \vdash \mathrm{e} = \mathrm{e}' : \tau}$

$$\frac{\mathrm{C} \vdash \Lambda\alpha{:}\kappa.\mathrm{e} : \forall\alpha{:}\kappa.\tau' \quad \mathrm{C} \vdash \tau : \kappa}{\mathrm{C} \vdash (\Lambda\alpha{:}\kappa.\mathrm{e})\,[\tau] = [\tau/\alpha]\,(\mathrm{e}) : [\tau/\alpha]\,(\tau')}$$

$$\frac{\mathrm{C} \vdash \mathbf{open}\ (\mathbf{pack}\ \tau\ \mathrm{e}\ \mathbf{as}\ \exists\alpha{:}\kappa.\tau')\ \mathbf{as}\ \alpha{:}\kappa, \mathrm{x}{:}\tau'\ \mathbf{in}\ \mathrm{e}' : \tau''}{\mathrm{C} \vdash \mathbf{open}\ (\mathbf{pack}\ \tau\ \mathrm{e}\ \mathbf{as}\ \exists\alpha{:}\kappa.\tau')\ \mathbf{as}\ \alpha{:}\kappa, \mathrm{x}{:}\tau'\ \mathbf{in}\ \mathrm{e}' = [\mathrm{e}/\mathrm{x}]\,([\tau/\alpha]\,(\mathrm{e}')) : \tau''}$$

Figure 2.14: Adding Second-Order Quantification.

and

$$\vdash \mathit{id}\ [\mathbf{bool}] : \mathbf{bool} \to \mathbf{bool}.$$

Similarly, we can define a function that, when applied to three type arguments, returns a composition function for functions on those types:

$$\begin{gathered}\vdash \Lambda\alpha{:}\star.\Lambda\beta{:}\star.\Lambda\gamma{:}\star.\lambda \mathrm{f}{:}\beta \to \gamma.\lambda \mathrm{g}{:}\alpha \to \beta.\lambda \mathrm{x}{:}\alpha.\mathrm{f}\ (\mathrm{g}\ \mathrm{x}) : \\ \forall\alpha{:}\star.\forall\beta{:}\star.\forall\gamma{:}\star.(\beta \to \gamma) \to (\alpha \to \beta) \to (\alpha \to \gamma)\end{gathered}$$

From a programming language perspective, adding universal quantification allows us to express *polymorphic* programs. A polymorphic program is a program whose operation is *generic* in a type. The operation of appending two lists is a good example of a polymorphic operation, since it is independent of the type of elements stored in the list. With simple types, we have to define a new, but essentially identical, append operation for each type of list element. Polymorphism allows us to get away with a *single* definition, leading to substantial savings in code and maintenance.

Example 2.2.4 *(Programming Examples).* With the ability to declare universally quantified types we can replace the infinite sets of constants $\mathbf{ifzero}_\tau$ and $\mathbf{fix}_\tau$ by two *polymorphic* constants **ifzero** and **fix**:

$$\frac{\vdash \mathrm{C}\ \mathbf{valid}}{\mathrm{C} \vdash \mathbf{ifzero} : \forall\alpha{:}\star.\mathbf{int} \to \alpha \to \alpha \to \alpha}$$

$$\frac{\mathrm{C} \vdash \mathbf{ifzero}\ [\tau]\ \bar{0}\ \mathrm{e}\ \mathrm{e}' : \tau}{\mathrm{C} \vdash \mathbf{ifzero}\ [\tau]\ \bar{0}\ \mathrm{e}\ \mathrm{e}' = \mathrm{e} : \tau}$$

$$\frac{\mathrm{C} \vdash \mathbf{ifzero}\ [\tau]\ \bar{i}\ \mathrm{e}\ \mathrm{e}' : \tau \quad i \not\equiv 0}{\mathrm{C} \vdash \mathbf{ifzero}\ [\tau]\ \bar{i}\ \mathrm{e}\ \mathrm{e}' = \mathrm{e}' : \tau}$$

$$\frac{\vdash \mathrm{C}\ \mathbf{valid}}{\mathrm{C} \vdash \mathbf{fix} : \forall\alpha{:}\star.(\alpha \to \alpha) \to \alpha}$$

$$\frac{\mathrm{C} \vdash \mathbf{fix}\ [\tau]\ \mathrm{e} : \tau}{\mathrm{C} \vdash \mathbf{fix}\ [\tau]\ \mathrm{e} = \mathrm{e}\ (\mathbf{fix}\ [\tau]\ \mathrm{e}) : \tau}$$

Moreover, we can exploit the polymorphism of **fix** and **ifzero** to give a *polymorphic* definition of iteration:

$$\begin{aligned}&\mathit{iter} \quad \equiv \\ &\quad \Lambda\alpha{:}\star.\lambda \mathrm{b}{:}\alpha.\lambda \mathrm{f}{:}\alpha \to \alpha. \\ &\qquad \mathbf{fix}\ [\mathbf{int} \to \alpha]\ (\lambda \mathrm{I}{:}\mathbf{int} \to \alpha.\lambda \mathrm{i}{:}\mathbf{int}.\mathbf{ifzero}\ [\alpha]\ \mathrm{i}\ \mathrm{b}\ (\mathrm{f}\ (\mathrm{I}\ (+\ \mathrm{i}\ (-\ \bar{1})))))\end{aligned}$$

It is easy to derive:

$$\vdash \mathit{iter} : \forall\alpha{:}\star.\alpha \to (\alpha \to \alpha) \to (\mathbf{int} \to \alpha)$$

We can now group the definition of zero, successor and polymorphic iteration into a record providing an implementation of the natural numbers:

$$\mathit{IntNat} \quad \equiv \quad \{\mathbf{z} = \bar{0}, \mathbf{s} = \mathit{succ}, \mathbf{i} = \mathit{iter}\}.$$

If we define the syntactic abbreviation:

$$\mathit{NAT}(\tau) \quad \equiv \quad \{\mathbf{z} : \tau, \mathbf{s} : \tau \to \tau, \mathbf{i} : \forall\alpha{:}\star.\alpha \to (\alpha \to \alpha) \to (\tau \to \alpha)\},$$

then we can show that:

$$\vdash \mathit{IntNat} : \mathit{NAT}(\mathbf{int}).$$

Finally, we can define something like a functor on records of type $\mathit{NAT}(\alpha)$, for any α:

$$\mathit{AddFun} \quad \equiv \quad \Lambda\alpha{:}\star.\lambda \mathrm{X}{:}\mathit{NAT}(\alpha).\{\mathbf{N} = \mathrm{X}, \mathbf{add} = \lambda \mathrm{n}{:}\alpha.\lambda \mathrm{m}{:}\alpha.\mathrm{X}.\mathbf{i}\ [\alpha]\ \mathrm{n}\ \mathrm{X}.\mathbf{s}\ \mathrm{m}\}.$$

For *AddFun*, we have:

$$\vdash \mathit{AddFun} : \forall\alpha{:}\star.\mathit{NAT}(\alpha) \to \{\mathbf{N} : \mathit{NAT}(\alpha), \mathbf{add} : \alpha \to \alpha \to \alpha\}.$$

We stress that $\mathit{NAT}(\tau)$ is a meta-level type abbreviation: it is not part of the syntax of the calculus.

Existential quantification, on the other hand, allows us to make the type of a term *abstract* by hiding some of the structure of its type. The term **pack** τ e **as** $\exists\alpha{:}\kappa.\tau'$ pairs the type τ with the term e and introduces an existential quantifier. Its type is derived from e's type, by hiding occurrences of the specified type τ according to the template $\exists\alpha{:}\kappa.\tau'$. The template is necessary to indicate which occurrences of τ (i.e. those marked by α) are to be hidden, and which are to remain visible. We can think of τ as the *witness* to the existentially quantified type variable, and e as the *term component* of the compound term.

The term **open** e **as** $\alpha{:}\kappa, \mathrm{x}{:}\tau$ **in** e$'$ eliminates an existential quantifier. Assume e has the existential type $\exists\alpha{:}\kappa.\tau$. Opening e provides the term e$'$ with access to the witness and term component of e. The type τ' of the entire phrase is the type of e$'$. To ensure the witness remains hidden, e$'$ may assume no more about it than that it is simply *some* type α of kind κ, and that the term component is a term x of type τ (mentioning α): the witness is *hypothetical*. The premise $\mathrm{C} \vdash \tau' : \star$ ensures that the hypothetical witness does not escape its scope by appearing in the result type τ': this guarantees that the type of the complete phrase is independent of the actual witness to α.

As a programming construct, existential types correspond to *abstract data types* [MP88]. Informally, an abstract data type consists of a set with a hidden representation and one or more operations over that representation, packaged up in a way that limits access to the representation according to some interface. Similarly, a term e of type $\exists\alpha{:}\kappa.\tau$ defines some type (the witness of α) with an operation (the term component of e) over this type. The type τ describes the interface of the abstract type to any client of e. Access to the abstract type is provided by opening e in the scope of the client. The operation e may, of course, be a record containing several operations, with its interface τ taking the form of a record type.

Example 2.2.5 *(Programming Example).* For instance, if we define:

$$\mathit{AbsNat} \quad \equiv \quad \textbf{pack int } \mathit{IntNat} \textbf{ as } \exists\beta{:}\star.\mathit{NAT}(\beta)$$

then, having quantified over all occurrences of **int** in the type of *IntNat*, we obtain:

$$\vdash \mathit{AbsNat} : \exists\beta{:}\star.\mathit{NAT}(\beta).$$

To use this term as an abstract implementation of the natural numbers, we first have to *open* it:

$$\begin{array}{l}\textbf{open } \mathit{AbsNat} \textbf{ as } \beta{:}\star, \mathrm{X}{:}\mathit{NAT}(\beta) \textbf{ in} \\ \qquad \textbf{pack } \beta \ \mathit{AddFun}\ [\beta]\ \mathrm{X} \textbf{ as } \exists\gamma{:}\star.\{\mathbf{N} : \mathit{NAT}(\gamma), \textbf{add} : \gamma \to \gamma \to \gamma\}\end{array}$$

This term evaluates to an abstract implementation of the naturals with addition. Since β is an ordinary type variable, denoting the witness of *AbsNat*, the fact that this witness is actually **int** is hidden from the client:

$$\textbf{pack } \beta \ \mathit{AddFun}\ [\beta]\ \mathrm{X} \textbf{ as } \exists\gamma{:}\star.\{\mathbf{N} : \mathit{NAT}(\gamma), \textbf{add} : \gamma \to \gamma \to \gamma\}.$$

Indeed, we can replace *AbsNat* by another abstract data type that uses a different representation of the naturals, without affecting the type of the complete program. In fact, we can replace *AbsNat* by *any* term of type $\exists\beta{:}\star.\mathit{NAT}(\beta)$ allowing the choice of representation to vary with the run-time value of that term.

Remark 2.2.3 *(Impredicativity).* In this system, quantified types are just ordinary types of kind $\star$. This means that we can write programs which, at run-time, choose between different implementations of an abstract data type or polymorphic function.

Type theories such as this, in which quantifying over some element of a *universe* (in this instance, the universe of types of kind $\star$) results in an element of the *same* universe, are called *impredicative.* This terminology distinguishes them from *predicative* theories in which quantification results in an element of a *higher* universe to which typically fewer operations apply.

2.2.3 Parameterisation over Types of a Kind

In Section 2.2.2, adding polymorphism meant extending the grammar of term phrases with terms parameterised by types, and terms applied to types. A rather different, in fact orthogonal, extension of the simply typed λ-calculus is to generalise the class of type phrases to allow *types* parameterised by types, $\Lambda\alpha{:}\kappa.\tau$, and applications of *types* to types, $\tau\ [\tau']$.

An intuitive example of a parameterised type is the generic type constant **list** that, when applied to an actual type argument τ, constructs the type **list** $[\tau]$. Intuitively, this type classifies terms that evaluate to lists of terms of type τ. This raises the question of what *kind* of type **list** is. The type constant **list** is a type phrase, but is not itself of kind $\star$, since it doesn't directly classify terms. Instead, we should think of **list** as a *function* on types: supplied with an argument τ of kind $\star$, it constructs a type of kind $\star$. To formalise this intuition, we extend the grammar of kinds with the function space $\kappa \to \kappa'$. We can then express the kind of **list** as the function space $\star \to \star$.

Figure 2.15 summarises the additional phrases and rules needed to extend the simply typed λ-calculus with parameterised types. Intuitively, these additions amount to turning the syntax of types into a simply typed λ-calculus, with types playing the role of "terms", and kinds playing the role of "types". The calculus at this level is a little simpler than the one in Section 2.2.1 since we only have a single "type" constant, the kind $\star$, and no "type" variables, but the ideas are the same. Since the equivalence of type phrases is no longer syntactic, but must take into account the notion of β-equivalence at the level of types, we also introduce the new equational judgement $\mathrm{C} \vdash \tau = \tau' : \kappa$ that formalises the equivalence between type phrases of the same kind. To make use of this equivalence, we require an additional term classification rule that allows us to view a term at syntactically different, but equivalent, types.

$\kappa \in$ Kind	$::=$	$\ldots$	
	$\mid$	$\kappa \rightarrow \kappa'$	function space
$\tau \in$ Type	$::=$	$\ldots$	
	$\mid$	$\Lambda\alpha{:}\kappa.\tau$	type parameterisation
	$\mid$	$\tau\ [\tau']$	type application

Type Classification $\boxed{\mathrm{C} \vdash \tau : \kappa}$

$$\frac{\mathrm{C}, \alpha : \kappa \vdash \tau : \kappa'}{\mathrm{C} \vdash \Lambda\alpha{:}\kappa.\tau : \kappa \rightarrow \kappa'}$$

$$\frac{\mathrm{C} \vdash \tau : \kappa' \rightarrow \kappa \quad \mathrm{C} \vdash \tau' : \kappa'}{\mathrm{C} \vdash \tau\ [\tau'] : \kappa}$$

Type Equivalence $\boxed{\mathrm{C} \vdash \tau = \tau' : \kappa}$

$$\frac{\mathrm{C}, \alpha : \kappa' \vdash \tau : \kappa \quad \mathrm{C} \vdash \tau' : \kappa'}{\mathrm{C} \vdash (\Lambda\alpha{:}\kappa'.\tau)\ [\tau'] = [\tau'/\alpha]\,(\tau) : \kappa}$$

(rules for congruence, symmetry, reflexivity and transitivity omitted)

Term Classification $\boxed{\mathrm{C} \vdash \mathrm{e} : \tau}$

$$\frac{\mathrm{C} \vdash \mathrm{e} : \tau \quad \mathrm{C} \vdash \tau = \tau' : \star}{\mathrm{C} \vdash \mathrm{e} : \tau'}$$

Figure 2.15: Adding Type Parameterisation.

Remark 2.2.4 *(The Logical Interpretation).* From a logical perspective, this extension alone is not particularly meaningful. However, if we combine it with the extension to quantified types, we obtain a system that corresponds to *higher-order* propositional logic. That is, we can now quantify not only over propositions, but also over functions from propositions to propositions and so on.

Example 2.2.6 *(Programming Examples).* Assuming the presence of quantified types, we can extend the calculus with parametric lists by adding the type constant **list** and the polymorphic term constants **nil**, **succ** and **listcase** with the following axioms.

$$\frac{\vdash \mathrm{C}\ \mathbf{valid}}{\mathrm{C} \vdash \mathbf{list} : \star \rightarrow \star}$$

$$\frac{\vdash \mathrm{C}\ \mathbf{valid}}{\mathrm{C} \vdash \mathbf{nil} : \forall\alpha{:}\star.\mathbf{list}\ [\alpha]}$$

$$\frac{\vdash \mathrm{C}\ \mathbf{valid}}{\mathrm{C} \vdash \mathbf{cons} : \forall\alpha{:}\star.\alpha \rightarrow (\mathbf{list}\ [\alpha]) \rightarrow (\mathbf{list}\ [\alpha])}$$

$$\frac{\vdash \mathrm{C}\ \mathbf{valid}}{\mathrm{C} \vdash \mathbf{listcase} : \forall\alpha{:}\star.\forall\beta{:}\star.(\mathbf{list}\ [\alpha]) \rightarrow \beta \rightarrow (\alpha \rightarrow (\mathbf{list}\ [\alpha]) \rightarrow \beta) \rightarrow \beta}$$

$$\frac{\mathrm{C} \vdash \mathbf{listcase}\ [\tau]\ [\tau']\ (\mathbf{nil}\ [\tau])\ \mathrm{e}\ \mathrm{e}' : \tau'}{\mathrm{C} \vdash \mathbf{listcase}\ [\tau]\ [\tau']\ (\mathbf{nil}\ [\tau])\ \mathrm{e}\ \mathrm{e}' = \mathrm{e} : \tau'}$$

$$\frac{\mathrm{C} \vdash \mathbf{listcase}\ [\tau]\ [\tau']\ (\mathbf{cons}\ [\tau]\ \mathrm{a}\ \mathrm{l})\ \mathrm{e}\ \mathrm{e}' : \tau'}{\mathrm{C} \vdash \mathbf{listcase}\ [\tau]\ [\tau']\ (\mathbf{cons}\ [\tau]\ \mathrm{a}\ \mathrm{l})\ \mathrm{e}\ \mathrm{e}' = \mathrm{e}'\ \mathrm{a}\ \mathrm{l} : \tau'}$$

In Example 2.2.4, we defined the shorthand $NAT(\tau)$ as an informal syntactic abbreviation at the meta-level; now we can express the equivalent parameterised type *within* the language:

$$NAT \quad \equiv \quad \Lambda\alpha{:}\star.\{\mathbf{z} : \alpha, \mathbf{s} : \alpha \rightarrow \alpha, \mathbf{i} : \forall\beta{:}\star.\beta \rightarrow (\beta \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta)\}.$$

and use it as object-level syntax. It is easy to show that $\vdash NAT : \star \rightarrow \star$.

Similarly, since we can parameterise over higher kinds (i.e. functional kinds), we can give an analogous specification of a package for polymorphic stacks, parameterised by the type constructor α implementing stacks:

$$\begin{aligned} \mathit{Stack} \quad \equiv \quad & \Lambda\alpha{:}\star \to \star. \\ & \{\mathbf{empty} : \forall\beta{:}\star.\alpha\ [\beta], \mathbf{push} : \forall\beta{:}\star.\beta \to (\alpha\ [\beta]) \to (\alpha\ [\beta]), \ldots\}. \end{aligned}$$

It is easy to show that $\vdash \mathit{Stack} : (\star \to \star) \to \star$. For instance, we can show that the naive implementation of stacks in terms of the list constructor has the type Stack $[\mathbf{list}]$:

$$\vdash \{\mathbf{empty} = \mathbf{nil}, \mathbf{push} = \mathbf{cons}, \ldots\} : \mathit{Stack}\ [\mathbf{list}].$$

Remark 2.2.5 *(The Effect on the Phase Distinction).* This extension does not affect the phase distinction, since none of the additional classification rules mention the term equivalence judgement. Although a type checker must do some non-trivial equational reasoning on types, the reasoning is still independent of the run-time equivalence of terms.

2.2.4 Subtypes

In the previous section, we extended the calculus with an equational theory on types, and added a rule allowing us to treat a term as having different, but equivalent types. Intuitively, equivalent type phrases classify equivalent collections of terms. A generalisation of this idea is to impose, not an equivalence relation, but a *pre-order* on types. Intuitively, if the pre-order $\mathrm{C} \vdash \tau \subseteq \tau' : \star$ holds (in context C), then the collection of terms classified by τ is a sub-collection of the terms classified by τ'. We say that τ is a *subtype* of τ'. To make use of subtyping, we need to add a new term classification rule, called the *subsumption* rule, that allows any term e of type τ to be used at type τ'. For subtyping to make sense, it must be the case that every operation on terms of type τ' is also defined on terms of the type τ. A more permissive interpretation of subtyping is to allow $\mathrm{C} \vdash \tau \subseteq \tau' : \star$ provided each term e in the collection τ can be *coerced* to a term in the collection τ' in some *coherent* manner[1]. This corresponds to allowing a non-trivial injection from τ into τ' rather than a simple inclusion of τ in τ'.

In theories with subtyping and functions, one typically adopts the following subtyping rule for function spaces. We say that $\tau_1 \to \tau_2$ is a subtype of $\tau_1' \to \tau_2'$ if, and only if, the domain τ_1' is a subtype of τ_1, and the range τ_2

[1]Roughly speaking, *coherence* means that the coercions resulting from different derivations of the same classification judgement are semantically equivalent.

SubTyping $\boxed{\mathrm{C} \vdash \tau \subseteq \tau' : \kappa}$

$$\frac{\mathrm{C} \vdash \alpha : \star}{\mathrm{C} \vdash \alpha \subseteq \alpha : \kappa}$$

$$\frac{\begin{array}{cc}\mathrm{C} \vdash \tau_1 \to \tau_2 : \star & \mathrm{C} \vdash \tau_1' \to \tau_2' : \star \\ \mathrm{C} \vdash \tau_1' \subseteq \tau_1 : \star & \mathrm{C} \vdash \tau_2 \subseteq \tau_2' : \star\end{array}}{\mathrm{C} \vdash \tau_1 \to \tau_2 \subseteq \tau_1' \to \tau_2' : \star}$$

Term Classification $\boxed{\mathrm{C} \vdash \mathrm{e} : \tau}$

$$\frac{\mathrm{C} \vdash \mathrm{e} : \tau \quad \mathrm{C} \vdash \tau \subseteq \tau' : \star}{\mathrm{C} \vdash \mathrm{e} : \tau'}$$

Figure 2.16: Adding Subtyping.

is a subtype of τ_2'. Observe that this definition is monotonic in the ranges, but anti-monotonic in the domains of the function spaces, which is why it is often called the *contravariant* rule. It is easy to motivate the contravariant rule if we view types as collections of terms. If every term in τ_1' also belongs to τ_1, then any function on terms in τ_1 is, less generally, a function on terms in τ_1'. Similarly, if every term in τ_2 also belongs to τ_2', then any function returning terms in τ_2 is, less generally, a function returning terms in τ_2'. Combining these ideas, we can say that the collection of functions with domain τ_1 and range τ_2 is a subcollection of the functions with domain τ_1' and range τ_2'.

Figure 2.16 shows the addition of subtyping to the simply typed λ-calculus, employing contravariant subtyping of function spaces and reflexive subtyping of type variables. Of course, this definition is trivial unless we add some specific subtyping judgements to get things started.

Remark 2.2.6 *(The Logical Interpretation).* If τ and τ' represent propositions in the context $\mathcal{C}$, then the judgement $\mathrm{C} \vdash \tau \subseteq \tau' : \star$ means that every proof of τ is also a proof of τ'.

Example 2.2.7 *(Record Subtyping).* A very natural idea is to construct a subtyping relation based on the structure of records. The intention is to allow the language to treat any record of type $\tau \equiv \{l_0 : \tau_0, \ldots, l_{m-1} : \tau_{m-1}\}$ as a record of type $\tau' \equiv \{l_0 : \tau_0, \ldots, l_{n-1} : \tau_{n-1}\}$, provided $m \geq n$, i.e. every field of τ' is declared with the same type in τ. Intuitively, this makes sense because a record of the wider type τ already supports all the field projections required of a record of the narrower type τ' (and then some). Informally, we can view τ as a subcollection of τ' if we interpret the record type $\{l_0 : \tau_0, \ldots, l_{n-1} : \tau_{n-1}\}$ as the collection of record terms containing *at least* the named components l_0 through l_{n-1} of the appropriate type. To axiomatise subtyping on records, we add record types as before together with the subtyping rule:

$$\frac{\begin{array}{ll}\mathrm{C} \vdash \{l_0 : \tau_0, \ldots, l_{m-1} : \tau_{m-1}\} : \star & \mathrm{C} \vdash \{l_0 : \tau'_0, \ldots, l_{n-1} : \tau'_{n-1}\} : \star \\ \forall i \in [n].\mathrm{C} \vdash \tau_i \subseteq \tau'_i : \star & \hfill m \geq n\end{array}}{\mathrm{C} \vdash \{l_0 : \tau_0, \ldots, l_{m-1} : \tau_{m-1}\} \subseteq \{l_0 : \tau'_0, \ldots, l_{n-1} : \tau'_{n-1}\} : \star}$$

This rule is slightly more general than we let on: it merely requires that the types of corresponding fields are in the subtype relation, without requiring them to be equivalent. Note also that the equivalence on record types means that the ordering of fields is arbitrary, so there is no requirement that the fields actually occur in the order we chose to present the rule.

Example 2.2.8 *(Programming Examples).* Suppose we adopt record subtyping and extend subtyping to quantified types by adding the covariant rule:

$$\frac{\mathrm{C} \vdash \forall\alpha{:}\kappa.\tau : \star \quad \mathrm{C} \vdash \forall\alpha{:}\kappa.\tau' : \star \quad \mathrm{C}, \alpha : \kappa \vdash \tau \subseteq \tau' : \star}{\mathrm{C} \vdash \forall\alpha{:}\kappa.\tau \subseteq \forall\alpha{:}\kappa.\tau' : \star}$$

and an analogous rule for the existential quantifier.

Then, continuing with our previous examples, consider the record *IntNat'*:

$$\begin{array}{l} \mathit{IntNat}' \equiv \\ \quad \{\mathbf{z} = \bar{0}, \mathbf{s} = \mathit{succ}, \mathbf{i} = \mathit{iter}, \mathbf{p} = \lambda \mathrm{x}{:}\mathbf{int}.\mathbf{ifzero}\ [\mathbf{int}]\ \mathrm{x}\ \bar{0}\ (+\ x\ (-\ \bar{1}))\} \end{array}$$

It generalises *IntNat* by declaring an additional field $\mathbf{p}$ containing the predecessor function. Without appealing to subtyping we have:

$$\begin{array}{l} \vdash \mathit{IntNat}' : \\ \quad \{\mathbf{z} : \mathbf{int}, \mathbf{s} : \mathbf{int} \to \mathbf{int}, \mathbf{i} : \forall\alpha{:}\star.\alpha \to (\alpha \to \alpha) \to (\mathbf{int} \to \alpha), \mathbf{p} : \mathbf{int} \to \mathbf{int}\}. \end{array}$$

Unfortunately, the application *AddFun* [**int**] *IntNat'* is ill-typed, since the function *AddFun* [**int**] does not expect an argument with a **p**-component.

However, with record subtyping, we have:

$$\begin{array}{l} \vdash \{\mathbf{z} : \mathbf{int}, \mathbf{s} : \mathbf{int} \to \mathbf{int}, \mathbf{i} : \forall\alpha{:}\star.\alpha \to (\alpha \to \alpha) \to (\mathbf{int} \to \alpha), \mathbf{p} : \mathbf{int} \to \mathbf{int}\} \\ \qquad \subseteq \{\mathbf{z} : \mathbf{int}, \mathbf{s} : \mathbf{int} \to \mathbf{int}, \mathbf{i} : \forall\alpha{:}\star.\alpha \to (\alpha \to \alpha) \to (\mathbf{int} \to \alpha)\} : \star \end{array}$$

Hence we can derive:

$$\vdash \mathit{IntNat}' : \{\mathbf{z} : \mathbf{int}, \mathbf{s} : \mathbf{int} \to \mathbf{int}, \mathbf{i} : \forall\alpha{:}\star.\alpha \to (\alpha \to \alpha) \to (\mathbf{int} \to \alpha)\}$$

and effectively ignore the existence of the additional **p**-component. It follows that the application *AddFun* [**int**] *IntNat′* is well-typed.

In fact, arguing rather differently, we can preserve the original type of *IntNat′* but employ the contravariant subtyping rule for function spaces. The idea is to reason that *AddFun* [**int**] also has the type of a function with the richer domain:

$$\{\mathbf{z} : \mathbf{int}, \mathbf{s} : \mathbf{int} \to \mathbf{int}, \mathbf{i} : \forall\alpha{:}\star.\alpha \to (\alpha \to \alpha) \to (\mathbf{int} \to \alpha), \mathbf{p} : \mathbf{int} \to \mathbf{int}\},$$

and then use this judgement to show that the application is well-typed.

Remark 2.2.7 *(The Effect on the Phase Distinction).* Adding subtypes does not affect the phase distinction. Although a type checker must do some non-trivial reasoning about subtypes, this reasoning is still independent of the run-time equational theory of terms.

2.2.5 First-Order Quantification

From a logical perspective, a natural generalisation of propositional logic is to extend the grammar of propositions with *predicates* on terms, and to permit the formation of new propositions by *first-order* universal and existential quantification over *terms*. This leads to first-order predicate logic. Figures 2.17 and 2.18 summarise the additions to the simply typed λ-calculus needed for a type-theoretic interpretation of predicates and first-order quantifiers. Types with first-order dependencies on terms are often called *dependent types* in the literature.

The idea is to view a predicate as a type phrase, that, when applied to some terms, yields a proposition. To do this, we generalise the grammar of type phrases with applications, τ e, of types to terms. To classify this new form of application, we also extend the structure of kinds with the function space $\tau \to \kappa$ that classifies type phrases mapping terms of type τ to types of kind κ.

For instance, if we view the type variable α as a collection of terms, then the kind $\alpha \to \star$ classifies the collection of unary predicates on α. Similarly, the kind $\alpha \to \alpha \to \star$ classifies the collections of binary predicates on α. If Φ is a predicate of kind $\alpha \to \star$, and a is a term of type α, then the application, Φ a, is the proposition that Φ holds for a.

$\kappa \in \mathrm{Kind}$	$::=$	$\ldots$	
	$\mid$	$\tau \rightarrow \kappa$	function space
$\tau \in \mathrm{Type}$	$::=$	$\ldots$	
	$\mid$	$\tau\ \mathrm{e}$	type application
	$\mid$	$\forall \mathrm{x}{:}\tau.\tau'$	universal quantification
	$\mid$	$\exists \mathrm{x}{:}\tau.\tau'$	existential quantification
$\mathrm{e} \in \mathrm{Term}$	$::=$	$\ldots$	
	$\mid$	$\langle \mathrm{e}, \mathrm{e}' \rangle\ \mathbf{as}\ \exists \mathrm{x}{:}\tau.\tau'$	pairing
	$\mid$	$\mathbf{fst}\ \mathrm{e}$	first projection
	$\mid$	$\mathbf{snd}\ \mathrm{e}$	second projection

Valid Kinds $\boxed{\mathrm{C} \vdash \kappa\ \mathbf{kind}}$

$$\frac{\mathrm{C} \vdash \tau : \star \quad \mathrm{C} \vdash \kappa\ \mathbf{kind}}{\mathrm{C} \vdash \tau \rightarrow \kappa\ \mathbf{kind}}$$

Type Classification $\boxed{\mathrm{C} \vdash \tau : \kappa}$

$$\frac{\mathrm{C} \vdash \tau : \tau' \rightarrow \kappa \quad \mathrm{C} \vdash \mathrm{e} : \tau'}{\mathrm{C} \vdash \tau\ \mathrm{e} : \kappa}$$

$$\frac{\mathrm{C}, \mathrm{x} : \tau \vdash \tau' : \star}{\mathrm{C} \vdash \forall \mathrm{x}{:}\tau.\tau' : \star}$$

$$\frac{\mathrm{C}, \mathrm{x} : \tau \vdash \tau' : \star}{\mathrm{C} \vdash \exists \mathrm{x}{:}\tau.\tau' : \star}$$

Figure 2.17: Adding First-Order Quantification.

Type Equivalence $\boxed{\mathrm{C} \vdash \tau = \tau' : \kappa}$

$$\frac{\mathrm{C} \vdash \tau : \tau' \to \kappa \quad \mathrm{C} \vdash \mathrm{e} = \mathrm{e}' : \tau'}{\mathrm{C} \vdash \tau\ \mathrm{e} = \tau\ \mathrm{e}' : \kappa} \qquad (=)$$

(other rules for congruence, symmetry, reflexivity and transitivity omitted)

Term Classification $\boxed{\mathrm{C} \vdash \mathrm{e} : \tau}$

$$\frac{\mathrm{C}, \mathrm{x} : \tau \vdash \mathrm{e} : \tau'}{\mathrm{C} \vdash \lambda \mathrm{x}{:}\tau.\mathrm{e} : \forall \mathrm{x}{:}\tau.\tau'}$$

$$\frac{\mathrm{C} \vdash \mathrm{e} : \forall \mathrm{x}{:}\tau'.\tau \quad \mathrm{C} \vdash \mathrm{e}' : \tau'}{\mathrm{C} \vdash \mathrm{e}\ \mathrm{e}' : [\mathrm{e}'/\mathrm{x}]\,(\tau)}$$

$$\frac{\mathrm{C} \vdash \exists \mathrm{x}{:}\tau.\tau' : \star \quad \mathrm{C} \vdash \mathrm{e} : \tau \quad \mathrm{C} \vdash \mathrm{e}' : [\mathrm{e}/\mathrm{x}]\,(\tau')}{\mathrm{C} \vdash \langle \mathrm{e}, \mathrm{e}' \rangle\ \mathbf{as}\ \exists \mathrm{x}{:}\tau.\tau' : \exists \mathrm{x}{:}\tau.\tau'}$$

$$\frac{\mathrm{C} \vdash \mathrm{e} : \exists \mathrm{x}{:}\tau.\tau'}{\mathrm{C} \vdash \mathbf{fst}\ \mathrm{e} : \tau}$$

$$\frac{\mathrm{C} \vdash \mathrm{e} : \exists \mathrm{x}{:}\tau.\tau'}{\mathrm{C} \vdash \mathbf{snd}\ \mathrm{e} : [\mathbf{fst}\ \mathrm{e}/\mathrm{x}]\,(\tau')}$$

$$\frac{\mathrm{C} \vdash \mathrm{e} : \tau' \quad \mathrm{C} \vdash \tau' = \tau : \star}{\mathrm{C} \vdash \mathrm{e} : \tau}$$

Term Equivalence $\boxed{\mathrm{C} \vdash \mathrm{e} = \mathrm{e}' : \tau}$

$$\frac{\mathrm{C} \vdash \lambda \mathrm{x}{:}\tau'.\mathrm{e} : \forall \mathrm{x}{:}\tau'.\tau \quad \mathrm{C} \vdash \mathrm{e}' : \tau'}{\mathrm{C} \vdash (\lambda \mathrm{x}{:}\tau'.\mathrm{e})\ \mathrm{e}' = [\mathrm{e}'/\mathrm{x}]\,(\mathrm{e}) : [\mathrm{e}'/\mathrm{x}]\,(\tau)}$$

$$\frac{\mathrm{C} \vdash \mathbf{fst}\ (\langle \mathrm{e}, \mathrm{e}' \rangle\ \mathbf{as}\ \exists \mathrm{x}{:}\tau.\tau') : \tau}{\mathrm{C} \vdash \mathbf{fst}\ (\langle \mathrm{e}, \mathrm{e}' \rangle\ \mathbf{as}\ \exists \mathrm{x}{:}\tau.\tau') = \mathrm{e} : \tau}$$

$$\frac{\mathrm{C} \vdash \mathbf{snd}\ (\langle \mathrm{e}, \mathrm{e}' \rangle\ \mathbf{as}\ \exists \mathrm{x}{:}\tau.\tau') : \tau''}{\mathrm{C} \vdash \mathbf{snd}\ (\langle \mathrm{e}, \mathrm{e}' \rangle\ \mathbf{as}\ \exists \mathrm{x}{:}\tau.\tau') = \mathrm{e}' : \tau''}$$

Figure 2.18: Adding First-Order Quantification (cont.)

In order to classify functions from term to terms, we need to extend the notion of simple function space $\tau \rightarrow \tau'$ to the *dependent* function space $\forall \mathrm{x}{:}\tau.\tau'$. Consider the parameterised term $\lambda \mathrm{x}{:}\tau.\mathrm{e}$. Observe that, because types may contain terms, the actual type τ' of the body e may depend on the term parameter x. Universally quantifying over x in the type $\forall \mathrm{x}{:}\tau.\tau'$ of $\lambda \mathrm{x}{:}\tau.\mathrm{e}$ captures this dependency. Intuitively, the type $\forall \mathrm{x}{:}\tau.\tau'$ describes a collection of functions f that, when applied to an argument a in the collection τ, return a term in the collection $[\mathrm{a}/\mathrm{x}]\,(\tau')$. Correspondingly, the type of an application is obtained by *substituting* the actual argument in the body of the function's type. Of course, the choice of the notation is not accidental: a function f of type $\forall \mathrm{x}{:}\alpha.\Phi\ \mathrm{x}$ corresponds to a proof of the proposition $\forall \mathrm{x}{:}\alpha.\Phi\ \mathrm{x}$. Whenever we apply f to a term a of type α, it returns a proof of the proposition Φ a. We can view the non-dependent function space $\tau \rightarrow \tau'$ as a degenerate case of universal quantification if we adopt the notational abbreviation $\tau \rightarrow \tau'$ for $\forall \mathrm{x}{:}\tau.\tau'$ whenever x does not occur free in τ', i.e. whenever the dependency is vacuous. In this sense, the rules for classifying a function $\lambda \mathrm{x}{:}\tau.\mathrm{e}$ and an application e e′ of Figure 2.17 generalise the corresponding rules of Figure 2.13, which may now be removed.

We can also add the first-order existential quantifier $\exists \mathrm{x}{:}\tau.\tau'$. It helps to consider the special case where the proposition τ' is an applied predicate Φ x. Intuitively, a (constructive) proof of the proposition $\exists \mathrm{x}{:}\tau.\Phi\ \mathrm{x}$ is a dependent pair $\langle \mathrm{a}, \mathrm{e}\rangle$ **as** $\exists \mathrm{x}{:}\tau.\Phi\ \mathrm{x}$, consisting of a witnessing term a of type τ, together with a proof e of the proposition Φ a. The template $\exists \mathrm{x}{:}\tau.\Phi\ \mathrm{x}$ is needed to indicate which occurrences of a in the type of e (i.e. those marked by x) are to be quantified. Access to the components of a pair is provided by the two term projections **fst** e and **snd** e. In the classification rule for **snd** e, the dependency of the second component's type on the quantified variable x is eliminated by substituting the first projection **fst** e for x. We can view the non-dependent cross product $\tau \times \tau'$ as a degenerate case of existential quantification if we adopt the notational abbreviation $\tau \times \tau'$ for $\exists \mathrm{x}{:}\tau.\tau'$, whenever x does not occur free in τ', i.e. whenever the dependency is vacuous.

Example 2.2.9 *(from Logic).* Consider the judgement:

$$\begin{array}{rl}\alpha : \star, \Phi : \alpha \rightarrow \star, \mathrm{x} : \alpha \ \ \vdash & \lambda \mathrm{f}{:}\forall \mathrm{y}{:}\tau.\Phi\ \mathrm{y}.\langle \mathrm{x}, \mathrm{f}\ \mathrm{x}\rangle\ \mathbf{as}\ \exists \mathrm{z}{:}\tau.\Phi\ \mathrm{z} : \\ & (\forall \mathrm{y}{:}\tau.\Phi\ \mathrm{y}) \rightarrow \exists \mathrm{z}{:}\tau.\Phi\ \mathrm{z}\end{array}$$

It corresponds to a proof of the proposition "if, for every y in α, Φ y is provable, then, for some z in α, Φ z is provable — provided α is a set, Φ is a predicate on α, and α is inhabited by x."

From a programming perspective, an intuitive example of a type depending on terms is the type of *lists of length* n, where n is a natural number. We could try to axiomatise such a type by adding constants $\mathbf{list}_\tau : \mathbf{nat} \rightarrow \star$, $\mathbf{nil}_\tau : \mathbf{list}_\tau\ \mathbf{zero}$ and $\mathbf{cons}_\tau : \forall \mathrm{n}{:}\mathbf{nat}.\tau \rightarrow \mathbf{list}_\tau\ \mathrm{n} \rightarrow \mathbf{list}_\tau\ (\mathbf{succ}\ \mathrm{n})$, for some fixed type of list elements τ. However, we will not bother to do this because it is easy to see that adding dependent types to a general-purpose programming language means that we have to abandon the compile-time/run-time phase distinction [Car88b, HMM90].

Remark 2.2.8 *(The Effect on the Phase Distinction).* First, observe that adopting dependent types introduces a syntactic dependency of types on terms, in the sense that the grammar of type phrases is defined in terms of the grammar of term phrases. The dependency of types and terms is deeper than mere syntax, however. From the perspective of logic, we need to equate propositions that are equal up to the equivalence of their subterms. In particular, the type equivalence judgement must include the congruence rule (Rule (=) of Figure 2.18):

$$\frac{\mathrm{C} \vdash \tau : \tau' \rightarrow \kappa \quad \mathrm{C} \vdash \mathrm{e} = \mathrm{e}' : \tau'}{\mathrm{C} \vdash \tau\ \mathrm{e} = \tau\ \mathrm{e}' : \kappa}$$

This rule equates two different applications of the same predicate τ provided their term arguments are equivalent.

Altering the equivalence on types to remove the dependency on term equivalence leads to a queer logic. On the one hand, removing the dependency by abandoning Rule (=) altogether means distinguishing between the intuitively equivalent propositions *Even* $\bar{2}$ and *Even* $(+\ \bar{1}\ \bar{1})$. On the other hand, modifying the Rule (=) to ignore term equivalence, for instance, by weakening the second premise:

$$\frac{\mathrm{C} \vdash \tau : \tau' \rightarrow \kappa \quad \mathrm{C} \vdash \mathrm{e} : \tau' \quad \mathrm{C} \vdash \mathrm{e}' : \tau'}{\mathrm{C} \vdash \tau\ \mathrm{e} = \tau\ \mathrm{e}' : \kappa}$$

means that the intuitively distinct propositions *Even* $\bar{2}$ and *Even* $\bar{1}$ are identified. In a system with true dependent types, the notion of type equivalence *must* depend on the notion of term equivalence.

In first-order predicate logic, we typically require that every term denotes. In Type Theory, this corresponds to having an equational theory on terms that is *strongly normalising.* In strongly normalising theories, every well-typed term has a unique normal form. This property is important because it means that the equivalence of two terms can be decided by comparing their normal forms. In a theory with dependent types, if we abandon *strongly normalisation* of terms then, because of the dependency of type equivalence on term equivalence, not only do we sacrifice the decidability of term equivalence, but we also lose the decidability of type checking. This rules out the use of dependent types in any general-purpose, i.e. non-terminating, programming language, unless we also forego decidable type checking. In short, dependent types violate the phase distinction.

2.2.6 Strong Higher-Order Existentials

The elimination rule for the higher-order existential quantifier $\exists\alpha{:}\kappa.\tau$ of Section 2.2.2 merely allows us to assume the existence of a *hypothetical* witness for the quantified type component. By contrast, the first-order existential quantifier of Section 2.2.5 is equipped with a *stronger* elimination form: we can project the *actual* witness of the existential using the first projection **fst** e. It is possible to design a similar construct for the higher-order case. The idea is to replace the, so-called, *weak* existential type $\exists\alpha{:}\kappa.\tau$ by the *strong* existential type $\Sigma\alpha{:}\kappa.\tau$, supporting the type projection **Fst** e and the term projection **Snd** e.[2] Figure 2.19 summarises the additional phrases and rules. The practical motivation for doing this is that strong existentials allow us to pair a type with a term depending on this type, *without* hiding the identity of the type component. As with weak existentials, the identity of the witness remains hidden in the *type* of a pair; however, it can always be recovered by projecting the pair's first component.

Example 2.2.10 *(A Programming Example).* For instance, using the strong existential we can define:

$$\mathit{StrongNat} \quad \equiv \quad \langle \mathbf{int}, \mathit{IntNat}\rangle \ \mathbf{as}\ \Sigma\beta{:}\star.\mathit{NAT}(\beta)$$

then, having quantified over all occurrences of **int** in the type of *IntNat*, we obtain:

$$\vdash \mathit{StrongNat} : \Sigma\beta{:}\star.\mathit{NAT}(\beta).$$

Notice that the witness to β is not apparent from the type of *StrongNat*. However, by projecting the term component of *StrongNat* we obtain:

$$\vdash \mathbf{Snd}\ \mathit{StrongNat} : \mathit{NAT}(\mathbf{Fst}\ \mathit{StrongNat}),$$

which, by reducing the type projection **Fst** *StrongNat*, is equivalent to:

$$\vdash \mathbf{Snd}\ \mathit{StrongNat} : \mathit{NAT}(\mathbf{int}).$$

[2]Actually, it is perfectly possible to have both the weak and the strong existential type in the same type theory [Luo90].

$$
\begin{array}{rcll}
\tau \in \mathrm{Type} & ::= & \ldots & \\
 & | & \Sigma\alpha{:}\kappa.\tau & \text{existential quantification} \\
 & | & \mathbf{Fst}\ \mathrm{e} & \text{type projection} \\
\mathrm{e} \in \mathrm{Term} & ::= & \ldots & \\
 & | & \langle \tau, \mathrm{e}' \rangle\ \mathbf{as}\ \Sigma\alpha{:}\kappa.\tau' & \text{pairing} \\
 & | & \mathbf{Snd}\ \mathrm{e} & \text{term projection}
\end{array}
$$

Type Classification $\boxed{\mathrm{C} \vdash \tau : \kappa}$

$$\frac{\mathrm{C}, \alpha : \kappa \vdash \tau : \star}{\mathrm{C} \vdash \Sigma\alpha{:}\kappa.\tau : \star}$$

$$\frac{\mathrm{C} \vdash \mathrm{e} : \Sigma\alpha{:}\kappa.\tau}{\mathrm{C} \vdash \mathbf{Fst}\ \mathrm{e} : \kappa}$$

Type Equivalence $\boxed{\mathrm{C} \vdash \tau = \tau' : \kappa}$

$$\frac{\mathrm{C} \vdash \mathbf{Fst}\ (\langle \tau, \mathrm{e} \rangle\ \mathbf{as}\ \Sigma\alpha{:}\kappa.\tau') : \kappa}{\mathrm{C} \vdash \mathbf{Fst}\ (\langle \tau, \mathrm{e} \rangle\ \mathbf{as}\ \Sigma\alpha{:}\kappa.\tau') = \tau : \kappa}$$

(rules for congruence, symmetry, reflexivity and transitivity omitted)

Term Classification $\boxed{\mathrm{C} \vdash \mathrm{e} : \tau}$

$$\frac{\mathrm{C} \vdash \Sigma\alpha{:}\kappa.\tau' : \star \quad \mathrm{C} \vdash \tau : \kappa \quad \mathrm{C} \vdash \mathrm{e} : [\tau/\alpha]\,(\tau')}{\mathrm{C} \vdash \langle \tau, \mathrm{e} \rangle\ \mathbf{as}\ \Sigma\alpha{:}\kappa.\tau' : \Sigma\alpha{:}\kappa.\tau'}$$

$$\frac{\mathrm{C} \vdash \mathrm{e} : \Sigma\alpha{:}\kappa.\tau}{\mathrm{C} \vdash \mathbf{Snd}\ \mathrm{e} : [\mathbf{Fst}\ \mathrm{e}/\alpha]\,(\tau)}$$

$$\frac{\mathrm{C} \vdash \mathrm{e} : \tau \quad \mathrm{C} \vdash \tau = \tau' : \star}{\mathrm{C} \vdash \mathrm{e} : \tau'}$$

Term Equivalence $\boxed{\mathrm{C} \vdash \mathrm{e} = \mathrm{e}' : \tau}$

$$\frac{\mathrm{C} \vdash \mathbf{Snd}\ (\langle \tau, \mathrm{e} \rangle\ \mathbf{as}\ \Sigma\alpha{:}\kappa.\tau) : \tau'}{\mathrm{C} \vdash \mathbf{Snd}\ (\langle \tau, \mathrm{e} \rangle\ \mathbf{as}\ \Sigma\alpha{:}\kappa.\tau) = \mathrm{e} : \tau'}$$

Figure 2.19: Adding Strong Higher-Order Existentials

Unlike the definition of *AbsNat* using the weak existential, the actual implementation of β in terms of the type **int** is transparent as soon as we access the term component of *StrongNat*. However, in order to establish that $\vdash NAT(\mathbf{Fst}\ \mathit{StrongNat}) = NAT(\mathbf{int}) : \star$ we have to first equate the term *StrongNat* with a pair: only then can we access the type component **int**, using the rule for type equivalence in Figure 2.19. In this case, this involves only trivial equational reasoning on terms since the term *StrongNat* is *already* in the form of a pair; in general, however, the term that we are projecting from may be *arbitrary*, requiring non-trivial computation to bring it into the form of a pair. If the term merely reduces to a variable, then the first-order dependency of the type on this variable cannot be removed.

Because of its transparency, the strong existential fails to provide the secure data abstraction associated with weak existentials. What it does provide is a mechanism for *pairing* related types and terms.

From a logical perspective, adding strong higher-order existentials requires extreme care. For instance, combining strong existentials with impredicative polymorphism leads to an *inconsistent* theory. Roughly speaking, using the first projection, it is possible to show that the type $\Sigma\alpha{:}\star.\mathbf{triv}$, where **triv** is some trivial inhabited type, is isomorphic to the collection of *all* types, i.e. the collection $\star$. Moreover, since our formulation of the strong existential is impredicative, we also have $\vdash (\Sigma\alpha{:}\star.\mathbf{triv}) : \star$. Thus we essentially have a type of all types. Much as admitting a set of all set leads to an inconsistent set theory, admitting a type of all types leads to an inconsistent type theory [HH86, HM93]. In this case, the inconsistency follows from the existence of well-typed, but non-normalising, terms. By the same token, the term equivalence judgement is undecidable. Because we have types depending on terms (i.e. the type phrase **Fst** e), this means that type checking is undecidable too.

It is possible to formulate a consistent version of strong higher-order existentials if we adopt a *predicative* notion of quantification. Recall that, in set theory, we can avoid the paradoxes resulting from admitting a set of all sets by introducing a new form of collection called a *class*, and distinguishing between *small collections*, i.e. sets, and *large* collections, e.g. the class of all sets. Similarly, in type theory, we can avoid the above inconsistency by introducing a distinction between the *universe* of *small* types, i.e. types of kind $\star$, and a *second* universe of *large* types, inhabited by strong existentials. This is the approach adopted, for instance, in Luo's Extended Calculus of Constructions [Luo90].

However, from a programming perspective, introducing a universe distinction means that dependent pairs may no longer be manipulated as ordinary terms. If we use dependent pairs to model modules with transparent type components, predicativity rules out the possibility of having first-class modules. Moreover, adopting a predicative theory does not alter the fact that strong existentials violate the phase distinction because the dependency of type equivalence on term term equivalence remains.

2.2.7 Summary

In this section, we introduced a number of type-theoretic constructs that, in combination, allowed us to emulate many of the examples we used to present Modules. In this thesis, we shall argue that a structure corresponds to a record; that functors correspond to polymorphic functions; that abstracting a structure by a signature corresponds to introducing an existentially quantified type; that the generation of new types corresponds to a weak form of existential elimination; that signatures correspond to parameterised types; and finally, that structure enrichment corresponds to a form of record subtyping. One of the distinguishing features of our analogy is that we do not resort to the use of first-order dependent types, nor do we resort to the use of strong higher-order existentials. This means that we avoid the problems and limitations both these features pose when integrated with a general-purpose programming language. Indeed, the only reason for presenting these last two concepts is that they figure prominently in the existing type-theoretic accounts of Standard ML Modules and its recent rivals.

2.3 Related Work

Research related to this thesis can be divided naturally into three categories: type-theoretic approaches to modular programming, type-theoretic accounts of Standard ML Modules, and type-theoretic alternatives to Standard ML Modules. We shall discuss each of these in turn and finish with a section on miscellaneous related work. In Chapter 9 we will revisit some of this work to compare it with the results of this thesis.

2.3.1 Type-Theoretic Approaches to Modular Programming

Mitchell and Plotkin's SOL

In their seminal paper [MP88], Mitchell and Plotkin make the original connection between the informal notion of abstract data type and existential quantification over types. They recognise that specifications of abstract data types correspond to existential types, the creation of an abstract data type to existential introduction, and the use of an abstract data type to existential elimination. SOL, the type theory they use to illustrate their ideas, is the simply-typed λ-calculus extended with second-order quantification over types. (The name SOL is an abbreviation for the second-order logic that results from the combination of the features we discussed in Sections 2.2.1 and 2.2.2). Mitchell and Plotkin observe that the impredicativity of SOL means that abstract datatypes are first-class values, allowing the *run-time* construction and selection of different implementations of the *same* abstract datatype. The paper focuses on the issue of data abstraction, and does not directly address other desirable features of modules languages, notably mechanisms for name space control and subtyping on module interfaces.

Cardelli's Quest

Cardelli's language Quest [Car88a, Car89, CL91, Car91] is an early type-theoretic programming language designed explicitly for the construction of modular programs. Quest is equipped with a form of dependent record, which allows sequences of related type and term definitions to be treated collectively. Dependent records are essentially a generalisation of SOL's existential types with a novel elimination form: type and term components of named records are accessed by a restricted form of the dot notation rather than the more unwieldy **open** phrase. This use of the dot notation to eliminate existentials is studied further in Cardelli and Leroy's paper [CL90].

Quest is rather different from Standard ML. Although Quest's record terms are similar to structure bodies, type components of Quest records are invariably *abstract*. As a result, interpreting a structure as a Quest record fails to account for the transparency of the structure's type components. Similarly, interpreting a functor as a Quest function on dependent records does not capture the behaviour that an application of the functor will propagate the realisation of type components from the functor's actual argument to its result. Like SOL, Quest supports first-class modules and avoids Standard ML's stratification between Core and Modules.

MacQueen's DL

In an influential position paper, MacQueen [Mac86] criticises module languages in which the only facility for grouping related definitions of types and terms is provided by existential types. MacQueen's argument is that existential types do not give adequate support for modular programming because the abstraction afforded by existential quantification is *too* strong:

1. Opening the same existential term twice yields two unrelated hypothetical type witnesses.
2. When building interrelated modules communicating via types of a common submodule, the submodule must be opened in a scope encompassing all of its uses, conflicting with the conceptually hierarchical structure of the system.

As an alternative, MacQueen advocates the use of dependent types as a basis for modular programming, drawing inspiration from the novel adoption of dependent types in Burstall and Lampson's experimental modules language Pebble [BL84, BL88]. MacQueen sketches the language DL, which is presented as a "de-sugared" version of the Standard ML Modules language originally proposed by him in [HMM86]. DL exploits a combination of strong higher-order existentials and first-order quantified types and, similar to Standard ML, exhibits a stratification between the core and modules languages. In MacQueen's interpretation, a signature specifying a type component corresponds to a *strong* higher-order existential type. The definition of a core type component within a structure is modeled by pairing a type with a term modeling the remainder of the structure. The ability to project the actual type component from such a term is intended to reflect Standard ML's notion of *transparent* type definitions in structures. A signature specifying a value or submodule is captured by a first-order existentially quantified type. The definition of a core value or module component within a structure is modeled by pairing a term with a term modeling the remainder of the structure. A functor mapping structures to structures is modeled as a dependent function from dependent pairs to dependent pairs. The apparent dependency of a functor's result type on its actual argument is captured by first-order universal quantification of the formal argument over the function space's range. The standard elimination rule for dependent functions roughly accords with Standard ML's ability to propagate type realisations from the actual argument to the result of a functor application.

Despite these similarities, DL fails to model most of the other important features of Modules. It is impossible to specify type definitions in signatures, preventing the expression of shared type components. Components are accessed not by identifier but by positional notation. DL has no notion of subtyping (corresponding to signature matching), making it impossible to treat a structure as if it had a type declaring fewer components, a type differing in the order of the structure's components, or a type differing in the degree of abstraction. Finally, DL fails to account for generative type definitions: every type component is transparent and there is no primitive support for type abstraction.

2.3.2 Type-Theoretic Accounts of Standard ML Modules

Harper and Mitchell's XML

Harper and Mitchell's calculus XML [HM93] is an attempt to apply MacQueen's ideas [Mac86] to provide a type-theoretic semantics of Standard ML. Surprisingly, they make no effort to relate their calculus to the published semantics of Standard ML [MTH90]. As a model of Standard ML Modules, XML exhibits essentially the same successes and shortcomings of DL. The use of weak existential types to account for Standard ML's notion of type generativity is sketched, but not incorporated in the definition of XML. Furthermore, the meta-theoretical implications of using strong higher-order existential types lead Harper and Mitchell to conclude that the stratification of XML into a modules language and a core language is *necessary*, unless we choose to admit (i) divergent terms in the absence of explicit recursion, and (ii) undecidable type-checking. Transferring the properties of their model to Standard ML, they conclude that Standard ML's stratification between Core and Modules is a theoretical requirement, not an historical accident. They consequently rule out the possibility of extending Standard ML with first-class modules. However, the validity of this conclusion depends crucially on the adequacy of their model.

Harper, Mitchell and Moggi's HML

Harper, Mitchell and Moggi's calculus HML [HMM90] is presented as a further refinement of the calculus XML. A serious failing of DL and XML (already acknowledged in [HM93]) is the absence of a *phase distinction* between compile-time type checking and run-time execution. As explained in Sections 2.2.5 and 2.2.6, the standard formulation of dependent types and strong existentials, on which DL and XML are based, implies that type checking involves the testing of term equivalence. Real programming languages generally have undecidable theories of term equivalence, rendering type-checking of realistic extensions of DL and XML undecidable. Standard ML, on the other hand, does obey a phase distinction.

Harper, Mitchell and Moggi [HMM90] refine their model accordingly: every dependently typed term of XML is interpreted in HML as a "mixed-phase" entity consisting of a compile-time and a run-time part. Thus, a higher-order dependent pair is interpreted as a pair of a compile-time type component and a run-time term component, in the usual way. However, a first-order dependent pair of two subterms is interpreted non-standardly as a pair of a compile-time type, pairing the subterms' type components, and a run-time term, pairing the subterms' term components. Similarly, a dependent function is interpreted non-standardly as a pair of a compile-time type component, consisting of a parameterised type that constructs the type component of the function's result as a function of the type component of the argument; and a run-time term component, consisting of a term that computes the term component of the function result as a polymorphic function of *both* the type *and* the term component of the argument. This split interpretation of dependent types yields a natural phase distinction. However, it is achieved by adopting a *non-standard* equational theory for dependent terms and types.

In HML, Standard ML Modules can still be modeled using dependent types following the ideas of DL and XML, but without sacrificing the phase distinction. Nevertheless, HML retains the other shortcomings of DL and XML: there is no account of the ability to specify type sharing in signatures, the ability to access structure components by name rather than position, the notion of structure enrichment, and type generativity.

Although elegant and interesting in its own right as the foundation of a practical modules system, HML is only indirectly related to Standard ML. Again, no attempt is made to formally relate the existing semantics of Standard ML to HML .

2.3.3 Type-Theoretic Alternatives to Standard ML Modules

Harper & Lillibridge's Translucent Sums

Harper and Lillibridge [HL94] present a type-theoretic alternative to Standard ML. They introduce a new type-theoretic construct called a *translucent sum*. Translucent sum terms are similar to the structures of Standard ML and the dependent records of Cardelli's Quest. Components of sum terms are named and are accessed by the dot-notation. For soundness reasons, type projections may only be applied to a subset of sum terms called *values*. Values are canonical, or fully evaluated, sum terms.

Akin to signatures, in a translucent sum type a type component may be specified in one of two ways: either *opaquely*, by specifying its name and kind, or *transparently*, by specifying its name and concrete definition. The term *translucent* refers to the possibility of having both opaque and transparent declarations in the same sum type. The benefit of this approach is that it supports transparency, *without* sacrificing the phase distinction: provided a given type component of a sum term is declared transparently in the sum's type, then the implementation of that type component can be determined, not by reduction of the sum, as in DL and XML, but by simple inspection of the sum's type. Any type component declared opaquely in the sum's type is treated as abstract, yielding the same degree of abstraction as SOL's weak existential types. Values of sum type are special, in the sense that a value may always be given a fully transparent type by a special typing rule. This rule replaces every opaque declaration in the value's type by a transparent declaration that is defined as a projection from the value itself.

The calculus employs a subtyping relation on sum types that, in particular, treats transparent type components as subtypes of opaque components, but also incorporates a structural subtyping relation similar to record subtyping. In Standard ML terms, the former aspect of this relation allows one to view any *realisation* of a signature (modeled as a more transparent sum type) as a subtype of the original signature (modeled as a more opaque sum type); the latter allows one to view any *enrichment* of the signature (modeled as a wider sum type) as a subtype of that signature (modeled as a narrower sum type).

Functors are modeled by functions on elements of sum types. In general, the type of a function's body may mention, and thus propagate, type components projected from the function's argument. If any of these components is opaque, the dependency of the result type on the formal argument cannot be eliminated, and the type of the function must be expressed using first-order universal quantification.

However, to avoid substituting terms in types, the usual elimination rule for dependent functions is abandoned. Instead, a dependently typed function may only be applied to an actual argument if it can first be given a non-dependent supertype (using a covariant subtyping rule). If the argument is a term of sum type, this is only possible provided the implementation of every type component, that is defined in the argument and propagated by the function, is transparent. Unless the actual argument belongs to the restricted set of values, this may not be the case. As a result, the calculus has the rather unnatural behaviour that certain function applications fail to type check, even though the argument is in the function's domain.

Although the calculus has a phase distinction, because the subtyping relation is undecidable, type-checking is undecidable too.

The translucent sum calculus supports a natural notion of higher-order functor, albeit without the "fully transparent" behaviour desired by MacQueen and Tofte [MT94]. Roughly speaking, a higher-order functor is *fully transparent* if it can propagate any incidental argument-result type dependency inherent in an actual argument (itself a functor), even if this dependency is left *unspecified* (i.e. abstract) in the range of the formal argument. Forcing the programmer to specify a particular argument-result dependency to achieve transparency is not satisfactory, since it decreases the generality of the higher-order functor.

Harper and Lillibridge's proposal is ambitious in avoiding a distinction between core and modules. The aim is to obtain a *uniform* language with first-class modules. The intention is that the need for a separate core language is subsumed by enriching the modules language directly with computational mechanisms normally associated with the core. This is made more explicit in follow-on work by Harper and Stone [SH96, HS97], that describes an interpretation of Standard ML into a variant of the translucent sum calculus, extended directly with state and exceptions. These papers give an involved syntactic translation, which, in this author's opinion, do little to clarify the semantics of Standard ML presented in [MTH90, MTH96]. Furthermore, there is currently no proof[3] that this translation is faithful to the original semantics. The thesis of Lillibridge [Lil97] develops the meta-theory of a drastically simplified type theory that is presented as a kernel version of the translucent sum calculi underlying [HL94, SH96, HS97].

Leroy's Modules

Historically, the goal of designing a simple separate compilation scheme for Standard ML Modules, akin to Modula-2's mechanism [Wir88], has remained elusive. In particular, the naive scheme that identifies implementations of compilation units with curtailed structures, and interfaces of these units with their curtailing signatures, is unsatisfactory. This approach fails because a curtailing signature rarely captures its implementation's full typing properties. Since type-checking of compilation units is meant to proceed by relying solely on declared interfaces, the discrepancy between a unit's signature and its implementation's actual typing properties means that it is possible to give examples of well-formed monolithic programs that no longer type-check when decomposed into separate compilation units.

As we shall see in Chapter 3, Standard ML distinguishes between syntactic type phrases (e.g. signatures), and the *semantic objects*, or types,

[3]Given the size of both the source and target languages, there probably never will be.

actually used to type-check phrases of the language. In particular, the semantic object assigned to a curtailed structure typically reveals additional type information, beyond that contained in its curtailing signature.

Arguing that the discrepancy between syntactic and semantic types is the root obstacle to separate compilation, Leroy proposes an alternative to the semantics of Modules in [Ler94]. His approach is to eliminate the discrepancy by formulating a type theory that relies solely on syntactic type information, thus removing the need for semantic objects. Syntactically, his language is very similar to Modules, apart from three significant departures. First, to simplify the theory he removes structure sharing constraints, allowing him to ignore the interesting but rarely exploited notion of structure sharing present in the original version of Standard ML [MTH90]. Second, to cater for structures with transparent type components, he enriches the syntax of signatures to allow manifest type definitions as well as abstract specifications. These subsume the functionality of Standard ML's less expressive type sharing constraints, and enable him to give more precise syntactic descriptions of structure types. Finally, in Leroy's calculus it is impossible to merely curtail a structure by a signature: only abstractions are supported. The separate compilation problem is solved by identifying the implementation of a compilation unit with an abstracted structure, and the interface of this unit with its abstracting signature. The typing rules of the calculus ensure that the signature fully captures the typing properties of the abstracted structure.

Leroy independently arrives at a theory that is in many ways similar to that of Harper and Lillibridge [HL94]. His enriched notion of signature is analogous to a translucent sum type, since a signature may contain a mixture of *abstract* (cf. opaque), and *manifest* (cf. transparent) type components. Signatures are used directly as the types of structures. Standard ML's notions of realisation and enrichment are combined in a single subtyping relation on signatures. Structure components are accessed by the dot-notation. However, the dot-notation is restricted and only applies to structures that are named by *paths*: a path is either a module identifier or the projection of a module identifier from a path. (The restriction to projections from paths is similar to, but stronger than, Harper and Lillibridge's restriction to projection from values.) The notion of path is significant, because a path can always be given a fully transparent signature by an operation called *strengthening*. Strengthening a path's signature redeclares any abstract type component in the signature as a manifest definition, expressed as a projection from the path itself.

Functors are modeled as functions taking structures to structures. As in

[HL94], first-order universal quantification must be used to describe the type of a functor whose result type mentions an abstract type component of its argument. Similarly, the calculus employs a non-standard elimination rule for dependent functions. The formulation of this rule varies slightly from one presentation of Leroy's calculus to another. To preserve the syntactic invariant of restricting projections to paths, in both [Ler94] and [Ler95], a dependent functor may only be applied if its argument is a path; if its argument is not a path, the functor must first be given a suitable *non-dependent* supertype (by using a covariant subtyping rule). Similar to Harper and Lillibridge's calculus [HL94], the calculi of both these papers have the rather unnatural behaviour that certain functor applications fail to type check, even though the type of the actual argument matches the signature of the functor's domain. Moreover, there appears to be no principled way of choosing between different non-dependent subtypes of a functor: as a result, these calculi fail to enjoy the principal (i.e. minimal) typing property. Another variant of these calculi, presented in [Ler96b], adopts a restricted grammar that only allows applications of functors to paths. Although this restriction seems to avoid the problem with principal types, it fails to capture Standard ML's ability to apply functors to *anonymous* arguments.

In [Ler96b], Leroy proves an equivalence between his notion of type abstraction, relying on syntactic signatures, and Standard ML's notion of type generativity. The equivalence result only holds for a restricted grammar of Standard ML programs. To circumvent this restriction, Leroy specifies a rewriting relation that transforms arbitrary well-typed Standard ML programs into well-typed programs belonging to this reduced grammar. However, the cost of this translation is to provide access to structures (though not type identities) that were anonymous, and thus inaccessible, in the original Standard ML source. In the same paper, Leroy gives a proof showing that his modified syntax for signatures elegantly subsumes the functionality of the type sharing constraints used in the original version of Standard ML [MTH90]. This is a significant result, since Leroy's syntax dispenses with the complicated, unification based mechanism needed to resolve sharing constraints. The syntax is compatible with Standard ML's elaboration to semantic objects, and has essentially been adopted in the revised definition of Standard ML [MTH96]. We also adopt it in this thesis.

Although Leroy's calculi in [Ler94, Ler96b] support a natural notion of higher-order functor, like Harper and Lillibridge's system, they fail to have the "fully transparent" behaviour of higher-order functors desired by MacQueen and Tofte [MT94]. Building on his previous work, Leroy [Ler95] offers a partial solution to the full transparency problem. It relies on extending the syntax of paths to include the *application* of a (functor) path to an (argument) path, allowing type components to be projected directly from functor applications, not merely from named structures. In this way, the type component resulting from the application of a functor to an argument can be expressed syntactically, provided both the functor and the argument are paths. The extension of paths is significant, because it allows a functor path to be given a fully transparent type by an extension of the *strengthening* operation. Strengthening a functor path's type redeclares any abstract type component in the functor's range as a manifest definition, expressed in terms of an application of the functor path to the bound argument of the functor's type. In combination, these extensions enable Leroy to approximate the fully transparent behaviour, provided programs adhere to the convention of only expressing functor applications involving paths. These extensions also yield a slightly different behaviour for functor application. Functor application is no longer *generative* but *applicative*, in the sense that two distinct applications of the same functor path to the same argument path result not in different, but equivalent, abstract types. The applicative behaviour actually provides better support for higher-order functors. In particular, it allows the programmer to specify sharing between abstract types returned by distinct occurrences of conceptually equivalent functor applications.

All of Leroy's calculi are defined with respect to an arbitrary core language. Like Standard ML, but unlike Harper and Lillibridge's system, Leroy's calculi maintain the rigid stratification between the core and modules. He only briefly mentions the possibility of first-class modules, and observes that removing the distinction between the core and modules is incompatible with applicative functors. Similar observations apply to Harper, Lillibridge and Stone's [HL94, Lil97, SH96, HS97] proposals: in these systems, the amalgamation of the core and modules languages means that the invariants needed to support applicative functors are violated. Leroy's stratification between core and modules ensures that both the subtyping relation on module types and typing relation on modules is decidable. Leroy's systems obey a phase distinction, and give a largely satisfactory, if somewhat restricted, treatment of most of Standard ML's features, bar one: in keeping with the syntactic treatment of module types, Leroy cannot account for the effect of merely curtailing, rather than abstracting, a structure by a signature.

Leroy's proposals have been implemented in a popular and robust compiler for a language very similar to Standard ML, Objective Caml [Ler97]. An accessible and almost literal implementation of Leroy's type system may be found in his tutorial introduction [Ler96a]. Pottier [Pot95] describes a prototype, full-scale implementation of Leroy's earliest calculus [Ler94] with higher-order, but non-transparent, functors. This work is interesting because it employs the "stamp-based" techniques of Standard ML, relying on the distinction between syntactic and semantic objects, to implement Leroy's calculus.

Despite their practical success, Leroy's proposals do have some theoretical weaknesses. The syntactic restrictions on paths, which are not closed under substitution of module terms for module identifiers, means that it is difficult to give a substitution-based dynamic semantics for his calculi. Courant [Cou97b] studies the failure of subject reduction for Leroy's module terms and proposes a variant calculus that admits a simple definition of module reduction. Courant's modifications allow him to state and prove a subject reduction theorem. His proposal, however, introduces a dependency of type checking on an equational theory of module terms. Although this approach blurs the phase distinction between compile-time and run-time, Courant carefully avoids using equational reasoning on core terms, preserving the decidability of type checking. Courant has managed to adapt his ideas to more general core languages including dependently typed *logics*[Cou97a]. However, the need to perform even limited compile-time equational reasoning on module terms is a distinct departure from the rigid phase-distinction enjoyed by Standard ML.

2.3.4 Miscellaneous Related Work

The second part of Tofte's thesis [Tof88] investigates a skeletal, first-order modules language. The language supports signatures, structures and functors but no core types or values. This is a kernel system for a forerunner of Standard ML's Modules. Tofte focuses on the meaning of structure identity and the problem of elaborating signatures containing structure sharing constraints to principal semantic objects. Aponté [Apo93] suggests a more modern, alternative treatment of the notion of structure identity and sharing actually adopted in Standard ML. In [Tof92, Tof93], Tofte describes preliminary work towards a direct extension of Standard ML with higher-order functors. In particular, he proves the existence of principal semantic objects for signatures that contain functor specifications. The research of Tofte and Aponté mainly concerns structure sharing, and is now obsolete due to the recent revision of Standard ML [MTH96], where this feature has been removed.

MacQueen and Tofte propose a "fully transparent" static semantics for higher-order functors in [MT94]. The behaviour of this static semantics is reflected in an early implementation of higher-order functors in the Standard ML of New Jersey compiler [AT 93]. The static semantics is very complicated and departs radically from the existing first-order semantics of Standard ML. In particular, it relies on the compile-time execution of a non-trivial language of identity stamps to account for full transparency. (On the other hand, the dynamic semantics for this proposal admits a straightforward formalisation that is studied by Maharaj and Gunter [MG93].)

Taking a different tack, Biswas [Bis95] proposes an alternative static semantics for higher-order functors that is also fully transparent. This semantics is based on a direct generalisation of a fragment of the existing Standard ML semantics. His ideas, which we believe have not received the attention they deserve, will be discussed in detail, reworked and extended to the full language in Chapter 5.

For a more radical but conceptually simpler approach to modules, based on interpreting some, but not all, of the features of Standard ML Modules directly in an extension of the Core language, see Jones [Jon96]. The companion paper by Nicklish and Peyton Jones [NJ96] offers an informal comparison of the two approaches.

2.3.5 Summary

One characteristic feature of Standard ML Modules is the ability to define structures containing both type and term components. Although the second-order existential types underlying SOL [MP88] and Quest [Car91] provide a similar facility, they cannot be used to model Standard ML structures: the realisation of a structure's type components is transparent, not abstract.

The primary motivation for the work on DL [Mac86] and XML [HM93] is to account for the transparency of type components by employing a familiar type-theoretic construct, the strong higher-order existential type. A necessary consequence of this decision, that introduces a dependency of types on terms, is the adoption of first-order existential types, to account for nested structures, and first-order universal types, to account for functors. The syntax of these constructs echoes the convenient syntax of Standard ML that allows type components to be expressed as projections from structure identifiers, and the type of a functor's body to refer to the functor's argument. DL and XML give an alternative account of structure transparency and the propagation of realisations across functor boundaries, while incorporating a natural notion of higher-order functor. Unfortunately, the reliance on strong existential types means that type equalities are established using term reduction, which runs foul of the phase-distinction. The fact that XML type checking becomes undecidable in the presence of *impredicative* strong existential types leads Harper and Mitchell to conclude that the extension of Standard ML with first-class modules is incompatible with decidable type checking. HML [HMM90], a further refinement of XML, is a predicative theory that preserves the phase distinction by adopting a non-standard formulation of dependent types.

Harper and Lillibridge [HL94] abandon the use of strong higher-order existentials altogether by proposing a new type-theoretic construct, the translucent sum, that models structures containing a mixture of opaque and transparent type components. The resulting modules calculus exploits first-order dependent types, but adopts non-standard elimination rules that ensure the phase distinction by propagating type equalities using subtyping instead of term reduction. The calculus fails to have the principal typing property because of the novel elimination rule for dependent functions. The calculus is impredicative, supporting first-class modules by removing the distinction, and thus the stratification, between core and modules language. Although equipped with higher-order modules, the identification of core and modules level computation means that neither applicative nor fully transparent functors can be accommodated without violating soundness. The decision not to distinguish between core and modules level types renders the subtyping relation, and thus typechecking, undecidable.

Leroy's calculi share many of the features of the translucent sum calculus. They do not enjoy the principal typing property due to a similar failing with the elimination rule for dependent functions. Leroy's proposals are more conservative in preserving the stratification between the core and modules languages. The stratification prohibits the use of first-class modules but has the advantage of maintaining a distinction between core and modules. By distinguishing between core and modules computation, Leroy can soundly accommodate an applicative and fully transparent semantics for higher-order functors, but this behaviour only applies to functor applications involving paths. Leroy's distinction between core and modules types ensures that the subtyping relation on module types is decidable.

One of the characteristics of Standard ML is that the static semantics of the language is defined, not in terms of the type syntax of the language, but with respect to an intermediate language of static semantic objects. During classification, type phrases are elaborated to semantic objects, and the classification of term phrases is done in terms of these semantic objects. This style of semantics can be criticised for two reasons. From a software engineering perspective, the classification of terms using semantic objects means that the type of a term cannot be reported to the programmer in the syntax of the language. In particular, this makes it more difficult to relate type errors to the source text of the program. It also raises the possibility that some terms have semantic objects that are not expressible in the syntax of the language, impeding simple approaches to separate compilation. From a proof engineering perspective, the classification of terms using semantic objects makes it difficult, and perhaps impossible, to prove the type soundness property of the language in terms of its type syntax. These properties are at odds with the syntactic nature of types and soundness proofs in type theory.

There are essentially two approaches to addressing this failing of Standard ML. The first is to reject the use of semantic objects to focus on the design of a modules language with a purely syntactic theory of types derived from the type syntax of Standard ML. Because of the inherent dependency of ML's type syntax on its term syntax, this leads naturally to type theories with dependent types, whose introduction and elimination rules must then be refined in order to obtain a phase distinction. The work discussed above is representative of this approach and has been successful in formulating concise, syntactic type systems whose meta-theories can be investigated using syntactic techniques adapted from type theory. The most recent proposals not only provide alternative treatments of Standard ML's features but also greatly extend them. Unfortunately, the meta-theoretic properties of the languages are less pleasing: we mention the absence of principal types for the calculi of Harper, Lillibridge and Leroy, and the need to blur the phase distinction of Leroy's calculus in order to prove subject reduction [Cou97b].

The second approach, the one taken in this thesis, is to retain the use of semantic objects, but to palliate their deficiencies by placing them on a more type-theoretic footing. This is particularly important because the current choice of semantic objects in Standard ML appears ad hoc and the definition of the static semantics too operational. The programmer's burden of understanding semantic objects can be eased, but not completely removed, by improving on the presentation of the static semantics, making type errors easier to report and understand. From the proof engineering perspective, it is still possible to prove a type soundness property for the language, but the property and its proof must be formulated with respect to semantic objects not syntactic types. Although less satisfactory than obtaining a purely syntactic description of the static semantics, we believe our approach to be acceptable because it offers other advantages. As we shall see, the use of semantic objects removes the necessity of dealing directly with first-order dependent types, by replacing first-order dependencies of types on terms by simpler, second- and higher-order dependencies of types on types. This makes the phase distinction clear from the outset and also avoids the syntactic restrictions that appear in the dependently typed systems (the restriction to projections from values in Harper and Lillibridges calculus, and the restriction to projections from paths in Leroy's). The use of semantic objects allows us to assign principal types to all terms, including those functor applications that lack principal types in the dependently typed theories of Harper, Lillibridge and Leroy. The semantics also scales naturally to higher-order functors that are both fully transparent and applicative in all cases, not just those involving applications between paths. Moreover, this semantics can be extended to include first-class modules without compromising the transparent and applicative behaviour of functors or, we believe, the decidability of modules subtyping. The latter result relies less on the fact that we employ semantic objects, and more on our decision to maintain the distinction between Core and Module, but drop the stratification, thus choosing the middle route between Leroy's stratified calculi and Harper and Lillibridge's amalgamation of the core and modules languages.

Chapter 3

The Static Semantics of Mini-SML

In this chapter, we set the scene for the remainder of this thesis by presenting the static semantics of a Modules and Core language in the style of Standard ML [MTH90, MTH96]. The two languages are presented separately. In Section 3.1, we present the Modules language. It models the main features of Standard ML's Modules language. Modules makes relatively few assumptions on the structure of the Core: our definition is parameterised by an arbitrary Core language. For concreteness, Section 3.2 presents a particular instance of the Core language: Core-ML. Although much simpler than Standard ML's Core, it nevertheless captures those features of the language that are relevant to the definition of Modules. In Section 3.3, we first define Mini-SML as the language obtained by combining the definitions of Modules and Core-ML. We then proceed with an informal analysis of the type-theoretic underpinnings of Mini-SML. Section 3.4 concludes the chapter with a brief summary.

3.1 Modules

In this section, we define the Modules language. Given a Core language supporting a notion of definable types and terms, Modules provides a typed calculus for manipulating collections of Core type and term definitions. Although the choice of Core language is largely arbitrary, we do need to make some assumptions on its structure. These are stated as hypotheses. They are sufficiently weak to accommodate a wide variety of Core languages.

k ∈ DefKind	kinds
d ∈ DefTyp	definable types
v ∈ ValTyp	value types

(a) Type Syntax

e ∈ ValExp	value expressions

(b) Term Syntax

Figure 3.1: Core Phrase Classes

3.1.1 Phrase Classes

We can present the syntax of Modules as a collection of *phrase classes* defined by a *grammar*. Modules is an explicitly typed language. For this reason, it is convenient to group the phrase classes of both Modules and the Core according to whether they belong to the syntax of types or the syntax of terms. The concrete grammar of Core definable types and terms depends on the Core language in question:

Hypothesis 3.1 (Core Phrase Classes).
We assume that the Core language defines a grammar for the four Core phrase classes shown in Figure 3.1.

- *Phrases* k ∈ DefKind *are the* kinds *used to specify Core definable types.*
- *Phrases* d ∈ DefTyp *are the* definable types *used to define the meaning of type identifiers.*
- *Phrase* v ∈ ValTyp *are the* value types *used to specify value identifiers.*
- *Phrases* e ∈ ValExp *are the* value expressions *used to define the meaning of value identifiers.*

To motivate Hypothesis 3.1 we can take a quick peek at Core-ML to see how it fits this generic description.

Example 3.1.1 *(Core-ML).* In Core-ML, a definable type is a *parameterised* simple type describing a family of simple types. The kind of a definable type is the number of type parameters it expects. A value type, on the other hand, is a *universally quantified* simple type, reflecting the polymorphism of Core-ML value expressions. Finally, a value expression is either a function, a function parameter, a function application, or an occurrence of a Core-ML value defined within a Module.

$t \in$ TypId	type identifiers
$x \in$ ValId	value identifiers
$X \in$ StrId	structure identifiers
$F \in$ FunId	functor identifiers

(a) Identifiers

$B \in$ SigBod	signature bodies
$S \in$ SigExp	signature expressions
$do \in$ TypOcc	type occurrences

(b) Type Syntax

$sp \in$ StrPath	structure paths
$b \in$ StrBod	structure bodies
$s \in$ StrExp	structure expressions
$vo \in$ ValOcc	value occurrences

(c) Term Syntax

Figure 3.2: Modules Phrase Classes

Remark 3.1.1. We distinguish between DefTyp and ValTyp specifically to accommodate languages like Core-ML. The two phrase classes may coincide in less complicated Core languages such as the simply typed λ-calculus.

We can now define the syntax of Modules. Figure 3.2 presents its phrase classes while Figure 3.3 defines their grammar.

We introduce type identifiers $t \in$ TypId to name Core definable types, and value identifiers $x \in$ ValId to name Core values. Identifiers $X \in$ StrId name structures and $F \in$ FunId functors. We shall assume TypId, ValId, StrId and FunId are pair-wise disjoint sets.

The type phrases of Modules are signature bodies and signatures. They specify the components of a structure by listing its component identifiers and their specifications.

The term phrases of the Modules language are structure paths, structure bodies and structure expressions. Structure paths provide access to structure identifiers and their sub-structures. Structure bodies are sequences of definitions binding type identifiers to Core definable types, value identifiers to Core values and structure identifiers to sub-structures. A functor defini-

TypId	$\overset{\text{def}}{=}$	$\{\mathbf{t}, \mathbf{u}, \ldots\}$	type identifiers
ValId	$\overset{\text{def}}{=}$	$\{\mathbf{x}, \mathbf{y}, \ldots\}$	value identifiers
StrId	$\overset{\text{def}}{=}$	$\{\mathbf{X}, \mathbf{Y}, \ldots\}$	structure identifiers
FunId	$\overset{\text{def}}{=}$	$\{\mathbf{F}, \mathbf{G}, \ldots\}$	functor identifiers
B	::=	**type** t = d; B	type definition
	|	**type** t : k; B	type specification
	|	**val** x : v; B	value specification
	|	**structure** X : S; B	structure specification
	|	ϵ_{B}	empty body
S	::=	**sig** B **end**	structure signature
do	::=	t	type identifier
	|	sp.t	type path
sp	::=	X	structure identifier
	|	sp.X	substructure projection
b	::=	**type** t = d; b	type definition
	|	**val** x = e; b	value definition
	|	**structure** X = s;b	structure definition
	|	**local** X = s **in** b	local structure definition
	|	**functor** F (X : S) = s **in** b	functor definition
	|	ϵ_{b}	empty body
s	::=	sp	structure path
	|	**struct** b **end**	structure body
	|	F s	functor application
	|	$\mathrm{s} \succeq \mathrm{S}$	signature curtailment
	|	$\mathrm{s} \setminus \mathrm{S}$	signature abstraction
vo	::=	x	value identifier
	|	sp.x	value path

Figure 3.3: Modules Grammar

tion introduces a named functor taking structures to structures. Structure expressions are phrases that evaluate to structures. They include paths, structure bodies, and functor applications as well as expressions that are curtailed or abstracted by a signature.

Let us examine the grammar of phrases to give an intuition of their semantics.

Signature expressions $\mathrm{S} \in \mathrm{SigExp}$ are encapsulated signature bodies. A signature body $\mathrm{B} \in \mathrm{SigBod}$ is a possibly empty sequence of type, value and structure specifications. Type specifications come in two forms. The phrase **type** $\mathrm{t} : \mathrm{k}; \mathrm{B}$ specifies a type component named t of kind k, without placing any further constraints on its actual *realisation* (i.e. its implementation). The phrase **type** $\mathrm{t} = \mathrm{d}; \mathrm{B}$, on the other hand, calls for t to be present and equal to the definable type d. The phrase **val** $\mathrm{x} : \mathrm{v}; \mathrm{B}$ specifies a value component named x of value type v. Finally, **structure** $\mathrm{X} : \mathrm{S}; \mathrm{B}$ specifies a structure component named X, matching (see below) the signature S. Subsequent phrases within a signature body may refer to previously specified identifiers.

A structure path $\mathrm{sp} \in \mathrm{StrPath}$ is a dot-separated, non-empty sequence of structure identifiers. The trivial path X accesses the structure bound to X in the current context. The phrase sp.X projects the structure component X from the enclosing structure sp.

A structure body $\mathrm{b} \in \mathrm{StrBod}$ is a possibly empty sequence of definitions. Each definition binds an appropriate identifier to a Core definable type, Core value, Modules structure or Modules functor. Subsequent phrases in the remainder of the body may refer to previously bound identifiers. Types, values and substructures introduced by phrases **type** $\mathrm{t} = \mathrm{d}; \mathrm{b}$, **val** $\mathrm{x} = \mathrm{e}; \mathrm{b}$, and **structure** $\mathrm{X} = \mathrm{s};\mathrm{b}$ become components of the encapsulating structure expression and can be accessed by the dot-notation. The phrase **local** $\mathrm{X} = \mathrm{s}$ **in** b, on the other hand, merely defines X for local use within b. Finally, the phrase **functor** $\mathrm{F}\ (\mathrm{X} : \mathrm{S}) = \mathrm{s}$ **in** b defines a functor F as a function on structures. X is the formal argument structure of F. The signature S specifies its type. The scope of the argument is the functor body s. The functor may be applied to any structure that matches the argument's signature S. Although the scope of the functor F is the remaining definitions in b, the functor itself does not become a component of the enclosing structure.

Structure expressions $\mathrm{s} \in \mathrm{StrExp}$ are phrases that evaluate to structures. They include structure paths as the means of referring to structures and substructures declared in the current context. The phrase **struct** b **end** encapsulates a structure body to form a structure. F s is the application of the functor F to an actual structure s. The phrase $s \succeq S$ matches the structure s against the signature S and curtails it accordingly: components of s not specified in S are no longer accessible from the curtailed structure. Nevertheless, those type components merely specified in S by **type** t : k phrases retain their actual realisation in s. The abstraction $s \setminus S$ is similar to the curtailment $s \succeq S$. However, the actual realisation of type components that are merely specified, but not defined, in S is *hidden* outside the abstraction, by generating new types for these components.

Informally, a structure expression *matches* a signature if it implements all of the components specified in the signature. In particular, the structure must *realise* all of the type components that are merely specified but not defined in the signature. Moreover, the structure must *enrich* the signature subject to this realisation: every specified type must be implemented by an equivalent type; every specified value must be implemented by a value whose type is at least as general as its specification; finally, every specified structure must be implemented by a structure that enriches its specification. The order in which components of the structure are actually defined is irrelevant. Furthermore, the structure is free to define more components than are specified in the signature.

We must, of course, equip the grammar of the Core language with a means of referring to the definitions of Core definable types and values introduced by Modules. There is little point in defining Modules otherwise. The phrase classes of type occurrences TypOcc and value occurrences ValOcc provide the syntactic interface that Modules presents to the Core.

Type occurrences are phrases that denote Core definable types. Conceptually, they belong to the syntax of types and are presumed to occur in the Core's type syntax. A type occurrence $do \in$ TypOcc is either a reference t to the definition of t in the current context, or the projection sp.t of the type component t from the structure path sp.

Value occurrences are phrases that evaluate to Core values. They belong to the syntax of terms, and are assumed to occur in the Core's term syntax. A value occurrence $vo \in$ ValOcc is either a reference x to the definition of x in the current context, or the projection sp.x of the value component x from the structure path sp.

Remark 3.1.2. In Standard ML, the phrase classes StrPath, TypOcc and ValOcc are called "long identifiers", since they boil down to dot-separated sequences of identifiers. In Chapter 5 we will do away with structure paths and generalise the dot-notation to apply to any Modules expression that evaluates to a structure. For this reason, we prefer the more neutral termi-

$d^{\mathrm{k}} \in \mathit{DefTyp}^{\mathrm{k}}$	definable types
$v \in \mathit{ValTyp}$	value types
$C \in \mathit{CoreContext}$	Core contexts

Figure 3.4: Core Semantic Objects

nology of type and value occurrences.

Remark 3.1.3. In Standard ML, functors may only be defined in a separate phrase class of "top level" definitions. To avoid introducing a further phrase class, we instead allow *strictly local* definitions of functors in structure bodies. In this way, outermost structure bodies can play the role of Standard ML's top-level. We stress that Modules remains first-order: functors may neither take functors as arguments nor return them as results.

3.1.2 Semantic Objects

Following Standard ML [MTH90, MTH96], the static semantics of Modules distinguishes between the type phrases of the language and the *semantic objects* they denote. As we shall see in Section 3.1.3, a type phrase is well-formed provided it denotes some semantic object, according to a denotation judgement. Similarly, a term phrase is well-typed provided it can be classified by some semantic object, according to a classification judgement. In this section, we define the semantic objects of Modules.

Notation. We will often re-use the same names for phrase classes and the semantic objects they denote. However, they are different. To avoid confusion, we will use roman font for a syntactic object $\mathrm{o} \in \mathrm{Object}$ and math italic for its semantic counterpart $o \in \mathit{Object}$.

Hypothesis 3.2 (Semantic Objects of the Core).
We assume that the Core language defines the following sets of semantic objects (summarised in Figure 3.4):

- *For each Core kind* $\mathrm{k} \in \mathrm{DefKind}$*, a set* $\mathit{DefTyp}^{\mathrm{k}}$ *of semantic definable types of kind* k*. We let* d^{k} *range over elements of the set* $\mathit{DefTyp}^{\mathrm{k}}$*. Semantic definable types are the denotations of well-formed type phrases* $\mathrm{d} \in \mathrm{DefTyp}$*.*

- *A set* ValTyp *of semantic value types, ranged over by* v*. Semantic value types are the denotations of well-formed value type phrases* $\mathrm{v} \in$

$\kappa \in \mathit{Kind}$	kinds classifying types
$\alpha^{\kappa} \in \mathit{TypVar}^{\kappa}$	type variables
$M,\ N,\ P,\ Q,\ R \in \mathit{TypVarSet}$	variable sets
$\nu^{\kappa} \in \mathit{TypNam}^{\kappa}$	type names
$\tau^{\kappa} \in \mathit{Typ}^{\kappa}$	types
$\mathcal{S} \in \mathit{Str}$	structures
$\mathcal{L} \in \mathit{Sig}$	signatures
$\mathcal{F} \in \mathit{Fun}$	functors
$\mathcal{C} \in \mathit{Context}$	contexts

Figure 3.5: Semantic Objects of Modules

ValTyp. *We also assume that they are the types used to classify value expressions* e ∈ ValExp*.*

- *A set CoreContext of Core contexts recording assumptions on Core-specific identifiers, as required to determine the denotations and classifications of Core phrases. We assume that Core contexts are finite maps whose domains are disjoint from the sets of type, value, structure and functor identifiers (*TypId, ValId, StrId *and* FunId*). The range of these finite maps is irrelevant to Modules and may vary according to the choice of Core language.*

Figures 3.5 and 3.6 summarise the *static semantic objects* assigned to module expressions. They serve the role of types in the type system of Modules. For convenience, whenever it aids the discussion, or helps to unambiguously abbreviate a number of related properties, we will use generic meta-variables o and $\mathcal{O}$ to range over more than one collection of semantic objects.

Definition 3.3 (Finite Sets and Maps). For sets A and B, Fin(A) denotes the set of *finite subsets* of A, and $A \overset{\text{fin}}{\to} B$ denotes the set of *finite maps* (partial functions with finite domain) from A to B. A finite map will often be written explicitly as a set in the form $\{a_1 \mapsto b_1, \cdots, a_k \mapsto b_k\}$, for $k \geq 0$. Let f and g be finite maps. Dom(f) and Rng(f) denote the domain of definition and range of f. The finite map $f+g$ has domain $\mathrm{Dom}(f)\cup\mathrm{Dom}(g)$ and values

$\kappa \in \mathit{Kind}$	$::=$	k	Core kind
$\alpha^{\kappa} \in \mathit{TypVar}^{\kappa}$	$\overset{\mathrm{def}}{=}$	$\{\alpha^{\kappa}, \beta^{\kappa}, \delta^{\kappa}, \gamma^{\kappa}, \ldots\}$	an infinite, denumerable set
$\alpha \in \mathit{TypVar}$	$\overset{\mathrm{def}}{=}$	$\biguplus_{\kappa \in \mathit{Kind}} \mathit{TypVar}^{\kappa}$	
$P, Q \in \mathit{TypVarSet}$	$\overset{\mathrm{def}}{=}$	$\mathrm{Fin}(\mathit{TypVar})$	
$\nu^{\kappa} \in \mathit{TypNam}^{\kappa}$	$::=$	α^{κ}	type variable
$\tau^{\kappa} \in \mathit{Typ}^{\kappa}$	$::=$	ν^{κ}	type name
	$\|$	d^{k} (where $\kappa \equiv \mathrm{k}$)	definable type
$\tau \in \mathit{Typ}$	$\overset{\mathrm{def}}{=}$	$\biguplus_{\kappa \in \mathit{Kind}} \mathit{Typ}^{\kappa}$	
$\mathcal{S} \in \mathit{Str}$	$::=$	$\mathrm{t} = \tau^{\mathrm{k}}, \mathcal{S}'$ (provided $\mathrm{t} \notin \mathrm{Dom}(\mathcal{S}')$)	type component
	$\|$	$\mathrm{x} : v, \mathcal{S}'$ (provided $\mathrm{x} \notin \mathrm{Dom}(\mathcal{S}')$)	value component
	$\|$	$\mathrm{X} : \mathcal{S}', \mathcal{S}''$ (provided $\mathrm{X} \notin \mathrm{Dom}(\mathcal{S}'')$)	structure component
	$\|$	$\epsilon_{\mathcal{S}}$	empty structure
$\mathcal{L} \in \mathit{Sig}$	$::=$	$(P)\mathcal{S}$	signature
$\mathcal{F} \in \mathit{Fun}$	$::=$	$(P)(\mathcal{S}, (Q)\mathcal{S}')$	functor
$\mathcal{C} \in \mathit{Context}$	$\overset{\mathrm{def}}{=}$	$\left\{ \begin{array}{c} C \cup \mathcal{C}_{\mathrm{t}} \cup \mathcal{C}_{\mathrm{x}} \cup \\ \mathcal{C}_{\mathrm{X}} \cup \mathcal{C}_{\mathrm{F}} \end{array} \middle\| \begin{array}{l} C \in \mathit{CoreContext}, \\ \mathcal{C}_{\mathrm{t}} \in \mathrm{TypId} \overset{\mathrm{fin}}{\to} \mathit{Typ}, \\ \mathcal{C}_{\mathrm{x}} \in \mathrm{ValId} \overset{\mathrm{fin}}{\to} \mathit{ValTyp}, \\ \mathcal{C}_{\mathrm{X}} \in \mathrm{StrId} \overset{\mathrm{fin}}{\to} \mathit{Str}, \\ \mathcal{C}_{\mathrm{F}} \in \mathrm{FunId} \overset{\mathrm{fin}}{\to} \mathit{Fun} \end{array} \right\}$	

Figure 3.6: Semantic Objects of Modules (cont.)

$(f+g)(a) \overset{\text{def}}{=}$ if $a \in \text{Dom}(g)$ then $g(a)$ else $f(a)$. If $\text{Rng}(g) \subseteq \text{Dom}(f)$ then $f \circ g$ is the finite map with domain $\text{Dom}(g)$ and values $(f \circ g)(a) \overset{\text{def}}{=} f(g(a))$. For $A \subseteq \text{Dom}(f)$, the *restriction* $f \downarrow A$ is the finite map with domain A and values $(f \downarrow A)(a) \overset{\text{def}}{=} f(a)$. Provided $\text{Dom}(f) \cap \text{Dom}(g) = \emptyset$ then the *parallel* map $f \mid g$ is the finite map $f \mid g \overset{\text{def}}{=} f \cup g$ (viewing f and g as sets) with domain $\text{Dom}(f) \cup \text{Dom}(g)$.

Notation (Enumerated Sets). We will occasionally need to enumerate the elements of a finite set. For k a natural number, we define $[k]$ to denote the finite set of indices $[k] \overset{\text{def}}{=} \{i \mid 0 \leq i < k\}$. Then $[0] = \emptyset$, the empty set of indices, and, for $k > 0$, $[k] = \{0, 1, \ldots, k-1\}$. We will often write $\{a_0, \ldots, a_{k-1}\}$ to enumerate the elements of a finite set of $k \geq 0$ elements, and use the notation a_i, for $i \in [k]$, to index the i-th element of this set.

As we shall see in Section 3.1.3, the effect of the definition **type** t = d is to extend the context with a declaration stating that the type identifier t has the denotation $d \in \mathit{DefTyp}$ of the phrase d ∈ DefTyp. A type specification **type** t : k, however, merely specifies an arbitrary realisation of the type identifier. Intuitively, the type denoted by t should represent an indeterminate definable type of kind k. Following Standard ML, we use *type names* $\nu \in \mathit{TypNam}$ to represent such indeterminates. In this chapter, a type name is just a kinded, second-order *type variable* $\alpha \in \mathit{TypVar}$, that ranges over semantic definable types of the appropriate kind. In order to record the denotation of a type identifier, allowing both the possibility that its denotation is a definable type, and the possibility that its denotation is just a type name, we also introduce an additional set of semantic objects: a *type* $\tau \in \mathit{Typ}$ is either a definable type or a type name. In this way, the possible denotations of a type identifier can be recorded uniformly as types. For instance, the declaration **type** t = d is recorded as t $= \tau$, where $\tau \equiv d$, while the specification **type** t : k is recorded as t $= \tau$, where $\tau \equiv \alpha$. Type variables, type names and types will be distinguished by their kind $\kappa \in \mathit{Kind}$, where the set Kind is constructed from the set of Core kinds DefKind. Formally, we define:

Definition 3.4 (Kinds, Type Variables, Type Names and Types).

Every kind $\kappa \in \mathit{Kind}$ is equivalent to some Core kind k ∈ DefKind used to specify definable types[1].

Kinds are used to index sets of kind-equivalent type variables, type names and types. For each kind $\kappa \in \mathit{Kind}$ we have:

[1]The distinction between DefKind and *Kind*, though vacuous here, will be exploited in Chapter 5, where we generalise the set *Kind*, keeping DefKind fixed.

- An infinite, denumerable set of *type variables*, $TypVar^{\kappa}$. A type variable $\alpha^{\kappa} \in TypVar^{\kappa}$ ranges over types in Typ^{κ}.
- A set of *type names*, $TypNam^{\kappa}$. A *type name* $\nu^{\kappa} \in TypNam^{\kappa}$ is just a type variable of kind κ.[2]
- A set of *types*, Typ^{κ}. A type $\tau^{\kappa} \in Typ^{\kappa}$ is either a type name or a Core definable type of the appropriate kind.

Hypothesis 3.5 (η-expansion of Type Names).
We assume that the Core provides a kind-preserving, injective function:

$$\eta(\_) \in \cup_{\mathrm{k} \in \mathrm{DefKind}} TypNam^{\mathrm{k}} \to DefTyp^{\mathrm{k}}$$

allowing any type name ν to be viewed as an equivalent *definable type $\eta(\nu)$ (of the same kind).*

Example 3.1.2 *(Core-ML's η).* In Core-ML, the type name α^2 is a variable for a definable type expecting two arguments. The Core-ML operation $\eta(\alpha^2)$ returns its η-expansion $\Lambda('a,'b).\alpha^2('a,'b)$.

The Core operation η, defined on type names, is extended to types:

Definition 3.6 ($\hat{\eta}$-expansion of Types). We define the operation $\hat{\eta}$ on types as follows:

$$\begin{array}{rcl} \hat{\eta}(\_) & \in & \cup_{\mathrm{k} \in \mathrm{DefKind}} Typ^{\mathrm{k}} \to DefTyp^{\mathrm{k}} \\ \hat{\eta}(\nu) & \stackrel{\mathrm{def}}{=} & \eta(\nu) \\ \hat{\eta}(d) & \stackrel{\mathrm{def}}{=} & d \end{array}$$

The operation $\hat{\eta}$ is used to define our notion of type equivalence:

Definition 3.7 (Equivalence of Types). We implicitly identify types with their $\hat{\eta}$-expansions. In particular, a type consisting of a type name is identified with the type consisting of the η-expansion of that name: we consider $\nu = \hat{\eta}(\nu)$. Moreover, we only consider as valid those equations between pairs of type variables, type names and definable types that compare objects of the same kind.

[2] Again, the distinction between sets $TypNam^{\kappa}$ and $TypVar^{\kappa}$, though vacuous here, will be exploited in Chapter 5, in which we generalise the sets $TypNam^{\kappa}$ to include constructs other than type variables.

Example 3.1.3 *(Core-ML's Equivalence of Types).* Continuing Example 3.1.2, for Core-ML, we consider that $\alpha^2 = \Lambda('a,'b).\alpha^2('a,'b)$, since the type on the right is an $\hat{\eta}$-expansion of the type on the left.

Definition 3.8 (Structures). A *semantic structure* $\mathcal{S} \in \mathit{Str}$ is a nested association list, associating identifiers with types, value types and semantic structures. Informally, semantic structures are the types assigned to structure bodies and structure expressions. They record the denotations of type components, and the types of value and structure components.

The *domain* of $\mathcal{S}$, written $\mathrm{Dom}(\mathcal{S})$, is the set of its components' identifiers:

$$\begin{array}{rcl}
\mathrm{Dom}(\_) & \in & \mathit{Str} \to \mathrm{Fin}(\mathrm{TypId} \cup \mathrm{ValId} \cup \mathrm{StrId}) \\
\mathrm{Dom}(\epsilon_{\mathcal{S}}) & \stackrel{\mathrm{def}}{=} & \emptyset \\
\mathrm{Dom}(\mathrm{t} = \tau, \mathcal{S}) & \stackrel{\mathrm{def}}{=} & \{\mathrm{t}\} \cup \mathrm{Dom}(\mathcal{S}) \\
\mathrm{Dom}(\mathrm{x} : v, \mathcal{S}) & \stackrel{\mathrm{def}}{=} & \{\mathrm{x}\} \cup \mathrm{Dom}(\mathcal{S}) \\
\mathrm{Dom}(\mathrm{X} : \mathcal{S}, \mathcal{S}') & \stackrel{\mathrm{def}}{=} & \{\mathrm{X}\} \cup \mathrm{Dom}(\mathcal{S}')
\end{array}$$

The provisos on structures in Figure 3.6 ensure that components are uniquely identified, allowing one to view a semantic structure as a triple of finite maps with corresponding (partial) retrieval functions:

$$\begin{array}{rcl}
\_(\_) & \in & (\mathit{Str} \times \mathrm{TypId}) \rightharpoonup \mathit{Typ} \\
\epsilon_{\mathcal{S}}(\mathrm{t}) & \stackrel{\mathrm{def}}{=} & \text{undefined} \\
(\mathrm{t}' = \tau, \mathcal{S})(\mathrm{t}) & \stackrel{\mathrm{def}}{=} & \left\{ \begin{array}{ll} \tau & \text{if } \mathrm{t} = \mathrm{t}' \\ \mathcal{S}(\mathrm{t}) & \text{otherwise.} \end{array} \right. \\
(\mathrm{x} : v, \mathcal{S})(\mathrm{t}) & \stackrel{\mathrm{def}}{=} & \mathcal{S}(\mathrm{t}) \\
(\mathrm{X} : \mathcal{S}, \mathcal{S}')(\mathrm{t}) & \stackrel{\mathrm{def}}{=} & \mathcal{S}'(\mathrm{t})
\end{array}$$

The retrieval functions for value and structure bindings are defined similarly.

Remark 3.1.4 *(Relating Semantic Structures to Record Types).* Semantic structures are very similar to *record types* (cf. Section 2.2.1). Like record types, they list the types of term components. Unlike record types, they also list the denotations of type components.

Notice that component identifiers are not variables, they are neither free nor bound but serve merely as tags, akin to the field names of record types. Contrast this with the nature of syntactic signatures that allow dependencies between successive components.

In fact, in Standard ML [MTH90, MTH96], semantic structures are *defined* as finite maps on identifiers: we prefer to use an inductive definition to make it easier to prove properties about them. The difference is not significant. In particular, our retrieval functions and the soon to be defined enrichment relation (Definition 3.17), are immune to the order in which components appear, as in Standard ML.

Definition 3.9 (Signatures). Signature expressions denote *semantic signatures* $\mathcal{L} \in \mathit{Sig}$. Every signature has the form $\mathcal{L} \equiv (P)\mathcal{S}$, where P is a set of type variables, and $\mathcal{S}$ is a semantic structure. Type variables in P are bound in $\mathcal{S}$, in the usual sense of free and bound variables.

Definition 3.10 (Functors). *Semantic functors* $\mathcal{F} \in \mathit{Fun}$ are the types assigned to functor identifiers. Every semantic functor has the form $\mathcal{F} \equiv (P)(\mathcal{S}, (Q)\mathcal{S}')$. Type variables in P are bound simultaneously in the functor *domain* $\mathcal{S}$ and the functor *range* $(Q)\mathcal{S}'$. Type variables in Q are bound in $\mathcal{S}'$.

Assume F is a functor of type $(P)(\mathcal{S}, (Q)\mathcal{S}')$. Informally, variables in P capture the type components of the domain $\mathcal{S}$ on which F is parametric; their possible occurrence in the range $(Q)\mathcal{S}'$ caters for the propagation of types from the functor's actual argument. Q is the set of *generative* or "new" type variables returned by an application of F. $\mathcal{S}'$ is the semantic structure of the result of the application.

Definition 3.11 (Contexts). A (Modules) *context* $\mathcal{C} \in \mathit{Context}$ is a finite map assigning semantic objects to identifiers. Note that, by our definition of contexts, every context determines a (possibly empty) Core context from which Core bindings can be retrieved. Moreover, adding a Core context binding to a Modules context result in a valid Modules context. In addition to Core level bindings, type, value, structure and functor identifiers are mapped to (semantic) types, value types, structures and functors respectively.

Contexts record the type information needed to type-check Modules and Core phrases and may be regarded as finite sets of assumptions: type identifiers are related to the types they denote; value, structure and functor identifiers are related to the types they inhabit. To stress the view of contexts as sets of assumptions, we define the following four operations for updating the context with either new or revised assumptions on identifiers:

$$\begin{array}{rcl}
\_[\_ = \_] & \in & (\mathit{Context} \times \mathrm{TypId} \times \mathit{Typ}) \to \mathit{Context} \\
\mathcal{C}[\mathrm{t} = \tau] & \stackrel{\mathrm{def}}{=} & \mathcal{C} + \{\mathrm{t} \mapsto \tau\} \\[2ex]
\_[\_ : \_] & \in & (\mathit{Context} \times \mathrm{ValId} \times \mathit{ValTyp}) \to \mathit{Context} \\
\mathcal{C}[\mathrm{x} : v] & \stackrel{\mathrm{def}}{=} & \mathcal{C} + \{\mathrm{x} \mapsto v\} \\[2ex]
\_[\_ : \_] & \in & (\mathit{Context} \times \mathrm{StrId} \times \mathit{Str}) \to \mathit{Context} \\
\mathcal{C}[\mathrm{X} : \mathcal{S}] & \stackrel{\mathrm{def}}{=} & \mathcal{C} + \{\mathrm{X} \mapsto \mathcal{S}\} \\[2ex]
\_[\_ : \_] & \in & (\mathit{Context} \times \mathrm{FunId} \times \mathit{Fun}) \to \mathit{Context} \\
\mathcal{C}[\mathrm{F} : \mathcal{F}] & \stackrel{\mathrm{def}}{=} & \mathcal{C} + \{\mathrm{F} \mapsto \mathcal{F}\}
\end{array}$$

Unlike semantic structures, contexts support re-bindings to identifiers. The definition of $\_ + \_$ ensures that subsequent bindings take precedence over previous ones.

Definition 3.12 (Free and Bound Type Variables). We let $\mathrm{FV}(\mathcal{O})$ denote the set of type variables occurring free in $\mathcal{O}$, where the notions of free and bound are defined as usual.

We implicitly identify semantic objects (i.e. signatures $(P)\mathcal{S}$ and functors $(P)(\mathcal{S}, (Q)\mathcal{S}')$) that are equivalent up to capture-avoiding, kind-preserving changes of bound type variables.

Definition 3.13 (Realisations).
A *realisation*:

$$\varphi \in \ \mathit{Real} \stackrel{\mathrm{def}}{=} \{f \in \mathit{TypVar} \stackrel{\mathrm{fin}}{\to} \mathit{Typ} \mid \forall\kappa.\forall\alpha^{\kappa} \in \mathrm{Dom}(f).f(\alpha^{\kappa}) \in \mathit{Typ}^{\kappa}\},$$

defines a kind-preserving, finite substitution of types for type variables.

Note that realisations are finite maps and we will often treat them as such.

For φ a realisation, we use the notation $\mathrm{Reg}(\varphi)$ to denote its *region*, i.e.

the set of type variables occurring free in its range:

$$\begin{array}{rcl}
\mathrm{Reg}(\_) & \in & \mathit{Real} \to \mathrm{Fin}(\mathit{TypVar}) \\
\mathrm{Reg}(\varphi) & \overset{\text{def}}{=} & \bigcup_{\tau \in \mathrm{Rng}(\varphi)} \mathrm{FV}(\tau) \\
& = & \bigcup_{\alpha \in \mathrm{Dom}(\varphi)} \mathrm{FV}(\varphi\,(\alpha)).
\end{array}$$

Hypothesis 3.14 (Realisation of Core Definable Types and Value Types). *We assume that the Core language is equipped with the following operations:*

$$\begin{array}{rcll}
\_\,(\_) & \in & (\mathit{Real} \times \mathit{DefTyp}^{\mathrm{k}}) \to \mathit{DefTyp}^{\mathrm{k}} & \textit{(for each } \mathrm{k} \in \mathrm{DefKind}\textit{)} \\
\_\,(\_) & \in & (\mathit{Real} \times \mathit{ValTyp}) \to \mathit{ValTyp} &
\end{array}$$

for applying realisations of type variables to definable types and value types.

Definition 3.15 (Realisation of Type Names and Types).
Realisation of type names and types is defined as follows:

$$\begin{array}{rcll}
\_\,(\_) & \in & (\mathit{Real} \times \mathit{TypNam}^{\kappa}) \to \mathit{Typ}^{\kappa} & (\text{for each } \kappa \in \mathit{Kind}) \\
\varphi\,(\alpha) & \overset{\text{def}}{=} & \begin{cases} \varphi(\alpha) & \text{if } \alpha \in \mathrm{Dom}(\varphi) \\ \alpha & \text{otherwise} \end{cases} &
\end{array}$$

and:[3]

$$\begin{array}{rcll}
\_\,(\_) & \in & (\mathit{Real} \times \mathit{Typ}^{\kappa}) \to \mathit{Typ}^{\kappa} & (\text{for each } \kappa \in \mathit{Kind}) \\
\varphi\,(d) & \overset{\text{def}}{=} & \varphi\,(d) & \\
\varphi\,(\nu) & \overset{\text{def}}{=} & \varphi\,(\nu) &
\end{array}$$

We extend realisations to structures, signatures and functors, avoiding capture of free variables by binding constructs (signatures $(P)\mathcal{S}$ and functors $(P)(\mathcal{S}, (Q)\mathcal{S}')$) in the usual way.

Modules defines a subtyping relation on semantic structures. If $\mathcal{S}$ is a subtype of $\mathcal{S}'$ then every phrase of type $\mathcal{S}$ may be used as a phrase of type $\mathcal{S}'$. Following Standard ML, the subtype relation is actually expressed as a subsumption relation, called *enrichment*. Thus $\mathcal{S}$ is a subtype of $\mathcal{S}'$ if, and

[3]The following definition looks circular, but it's not: the key is to read the left-hand side as a case analysis on the outer form of the semantic type, and to read the right-hand side as applying the realisation to the uncovered subterm.

Structure Enrichment $\boxed{\mathcal{S} \succeq \mathcal{S}'}$

$$\frac{\begin{array}{l}\mathrm{Dom}(\mathcal{S}) \supseteq \mathrm{Dom}(\mathcal{S}') \\ \forall \mathrm{t} \in \mathrm{Dom}(\mathcal{S}').\mathcal{S}(\mathrm{t}) = \mathcal{S}'(\mathrm{t}) \\ \forall \mathrm{x} \in \mathrm{Dom}(\mathcal{S}').\mathcal{S}(\mathrm{x}) \succeq \mathcal{S}'(\mathrm{x}) \\ \forall \mathrm{X} \in \mathrm{Dom}(\mathcal{S}').\mathcal{S}(\mathrm{X}) \succeq \mathcal{S}'(\mathrm{X})\end{array}}{\mathcal{S} \succeq \mathcal{S}'}$$

Figure 3.7: Enrichment

only if, $\mathcal{S}$ enriches $\mathcal{S}'$. It is the presence of subtyping that, on the one hand, allow us to apply a functor to an argument with a type that is richer than required; and, on the other hand, allows us to coerce the actual type of a structure expression to one that is less rich.

The definition of enrichment between structures extends a Core-dependent subtyping relation on value types:

Hypothesis 3.16 (Enrichment between Value Types).
We assume that the Core language is equipped with an enrichment *relation on value types:*

$$\_ \succeq \_ \in \mathit{ValTyp} \times \mathit{ValTyp}.$$

The relation is intended to capture a subtyping relation: provided $v \succeq v'$, read v enriches *v', then any Core value of type v may also be regarded as a value of type v'.*

We require that $\_ \succeq \_$ is a pre-order, i.e. that it is a reflexive and transitive relation.

We also require that $\_ \succeq \_$ is closed under realisation: that is, whenever $v \succeq v'$ then, for any realisation φ, we also have $\varphi(v) \succeq \varphi(v')$.

We can now introduce the enrichment relation on structures. Informally, $\mathcal{S}$ *enriches* $\mathcal{S}'$, written $\mathcal{S} \succeq \mathcal{S}'$, if and only if:

- $\mathcal{S}$ has at least the components of $\mathcal{S}'$;
- the type components common to both structures are equivalent;
- the type of every term component of $\mathcal{S}$ is at least as rich as the type of any corresponding component of $\mathcal{S}'$.

Formally, we define:

$\mathcal{C} \vdash \mathrm{d} \triangleright d$

In context $\mathcal{C}$, definable type d denotes definable type d.

$\mathcal{C} \vdash \mathrm{v} \triangleright v$

In context $\mathcal{C}$, value type v denotes value type v.

(a) Denotation Judgements

$\mathcal{C} \vdash \mathrm{e} : v$

In context $\mathcal{C}$, value expression e has value type v.

(b) Classification Judgements

Figure 3.8: Core Judgements

Definition 3.17 (Enrichment between Structures). The *enrichment* relation between structures is defined as the least relation:

$$\_ \succeq \_ \in \mathit{Str} \times \mathit{Str}$$

closed under the rule in Figure 3.7.

Note that the equalities in the second premise of this rule implicitly require that common type components have the same kind (recall Definition 3.7). Also, observe that the order in which components appear within $\mathcal{S}$ and $\mathcal{S}'$ is irrelevant to the definition of $\mathcal{S} \succeq \mathcal{S}'$.

Given Hypothesis 3.16, it is easy to show that $\_ \succeq \_$ is a pre-order that is closed under realisation.

Finally, *matching* a structure against a signature is a combination of realisation and enrichment:

Definition 3.18 (Signature Matching). A structure $\mathcal{S}$ *matches* a signature $\mathcal{L} \equiv (P)\mathcal{S}'$ if, and only if, there exists a realisation φ with $\mathrm{Dom}(\varphi) = P$ such that $\mathcal{S} \succeq \varphi(\mathcal{S}')$.

3.1.3 Static Semantics

In this section we define the judgements of the static (or type-checking) semantics of Modules. There are essentially two kinds of judgement. A *denotation judgement* has the form $\mathcal{C} \vdash \mathrm{p} \triangleright p$. It relates a type phrase p to the semantic object p that it denotes in the context $\mathcal{C}$. A *classification judgement* has the form $\mathcal{C} \vdash \mathrm{p} : o$. It relates a term phrase p to the semantic object o that it inhabits in the context $\mathcal{C}$. We can think of o as the type of p.

$\mathcal{C} \vdash \mathrm{B} \triangleright \mathcal{L}$

In context $\mathcal{C}$, signature body B denotes signature $\mathcal{L}$.

$\mathcal{C} \vdash \mathrm{S} \triangleright \mathcal{L}$

In context $\mathcal{C}$, signature expression S denotes signature $\mathcal{L}$.

$\mathcal{C} \vdash \mathrm{do} \triangleright d$

In context $\mathcal{C}$, type occurrence do denotes definable type d.

(a) Denotation Judgements

$\mathcal{C} \vdash \mathrm{sp} : \mathcal{S}$

In context $\mathcal{C}$, path sp has structure $\mathcal{S}$.

$\mathcal{C} \vdash \mathrm{vo} : v$

In context $\mathcal{C}$, value occurrence vo has value type v.

(b) Classification Judgements

$\mathcal{C}, N \vdash \mathrm{b} : \mathcal{S} \Rightarrow M$

In context $\mathcal{C}$ and state N, structure body b has structure $\mathcal{S}$, generating fresh types M.

$\mathcal{C}, N \vdash \mathrm{s} : \mathcal{S} \Rightarrow M$

In context $\mathcal{C}$ and state N, structure expression s has structure $\mathcal{S}$, generating fresh types M.

(c) Generative Classification Judgements

Figure 3.9: Modules Judgements

Hypothesis 3.19 (Core Judgements).

We assume the static semantic of the Core provides inference rules defining the judgements in Figure 3.8. Each judgement is accompanied by its English reading.

Remark 3.1.5. For Modules to be useful, we must allow Core type phrases to contain type occurrences $\mathrm{do} \in \mathrm{TypOcc}$ accessing type components of structures. Similarly, we must permit Core term phrases to contain value occurrences $\mathrm{vo} \in \mathrm{ValOcc}$ accessing value components. In particular, we shall allow the rules defining Core judgements to refer to the Modules judgements $\mathcal{C} \vdash \mathrm{do} \triangleright d$ and $\mathcal{C} \vdash \mathrm{vo} : v$ defined below. Since the denotations and types of these phrases must be determined in a Modules context $\mathcal{C} \in \mathit{Context}$, the Core judgements must also be defined with respect to Modules contexts. Recall that *Context* generalises the set *CoreContext* of Core-specific contexts.

The static semantics of Modules is defined by the judgements in Figure 3.9. Each judgement is accompanied by its English reading. In Figure 3.9, we encounter a third form of judgement: the *generative classification judgement* $\mathcal{C}, N \vdash \mathrm{p} : o \Rightarrow M$. Like an ordinary classification judgement, it relates a term phrase p to the semantic object o that it inhabits. However, the classification of the phrase is allowed to *generate* new type variables. N records the set of type variables existing *prior* to the classification of the phrase. M records the set of *new* type variables generated *during* the classification of the phrase.

The judgements of Modules are defined by the following inference rules.

Denotation Rules

Signature Bodies $\boxed{\mathcal{C} \vdash \mathrm{B} \triangleright \mathcal{L}}$

A signature body B denotes a semantic signature $\mathcal{L} \equiv (P)\mathcal{S}$. The bound variables of the signature arise from type specifications **type** $\mathrm{t} : \mathrm{k}$ appearing within the body and its sub-signatures (see Rule E-2 below). P is the set of types specified by B. Essentially, each component of B gives rise to a component of the structure $\mathcal{S}$, by replacing the components' specification by its denotation in the current context. The specification of a component is added to the context before determining the denotations of subsequent specifications. The side-conditions on bound type variables merely prevent capture of free variables. They can always be satisfied by suitable renamings. The remaining side-conditions ensure that components are uniquely identified.

$$\frac{\begin{array}{ll}\mathcal{C} \vdash \mathrm{d} \rhd d & P \cap \mathrm{FV}(d) = \emptyset \\ \mathcal{C}[\mathrm{t} = d] \vdash \mathrm{B} \rhd (P)\mathcal{S} & \mathrm{t} \notin \mathrm{Dom}(\mathcal{S})\end{array}}{\mathcal{C} \vdash \textbf{type}\ \mathrm{t} = \mathrm{d}; \mathrm{B} \rhd (P)\mathrm{t} = d, \mathcal{S}} \tag{E-1}$$

$$\frac{\mathcal{C}[\mathrm{t} = \alpha^{\mathrm{k}}] \vdash \mathrm{B} \rhd (P)\mathcal{S} \quad \alpha^{\mathrm{k}} \notin \mathrm{FV}(\mathcal{C}) \cup P \quad \mathrm{t} \notin \mathrm{Dom}(\mathcal{S})}{\mathcal{C} \vdash \textbf{type}\ \mathrm{t} : \mathrm{k}; \mathrm{B} \rhd (\{\alpha^{\mathrm{k}}\} \cup P)\mathrm{t} = \alpha^{\mathrm{k}}, \mathcal{S}} \tag{E-2}$$

$$\frac{\begin{array}{ll}\mathcal{C} \vdash \mathrm{v} \rhd v & P \cap \mathrm{FV}(v) = \emptyset \\ \mathcal{C}[\mathrm{x} : v] \vdash \mathrm{B} \rhd (P)\mathcal{S} & \mathrm{x} \notin \mathrm{Dom}(\mathcal{S})\end{array}}{\mathcal{C} \vdash \textbf{val}\ \mathrm{x} : \mathrm{v}; \mathrm{B} \rhd (P)\mathrm{x} : v, \mathcal{S}} \tag{E-3}$$

$$\frac{\begin{array}{ll}\mathcal{C} \vdash \mathrm{S} \rhd (P)\mathcal{S} & \\ \mathcal{C}[\mathrm{X} : \mathcal{S}] \vdash \mathrm{B} \rhd (Q)\mathcal{S}' & P \cap \mathrm{FV}(\mathcal{C}) = \emptyset \\ Q \cap (P \cup \mathrm{FV}(\mathcal{S})) = \emptyset & \mathrm{X} \notin \mathrm{Dom}(\mathcal{S}')\end{array}}{\mathcal{C} \vdash \textbf{structure}\ \mathrm{X} : \mathrm{S}; \mathrm{B} \rhd (P \cup Q)\mathrm{X} : \mathcal{S}, \mathcal{S}'} \tag{E-4}$$

$$\frac{}{\mathcal{C} \vdash \epsilon_{\mathrm{B}} \rhd (\emptyset)\epsilon_{\mathcal{S}}} \tag{E-5}$$

(E-1) The signature body specifies a type named t with the same denotation as the given definable type d.

(E-2) The signature body specifies a type component t of kind k with an indeterminate realisation. The variable α^{k}, which is chosen to be fresh with respect to the current context, is used to represent the indeterminate. Provided B denotes in the extended context, the variable α^{k} is discharged, added to the set P of types specified by B, and bound in the compound signature $(\{\alpha^{\mathrm{k}}\} \cup P)\mathrm{t} = \alpha^{\mathrm{k}}, \mathcal{S}'$.

(E-4) The signature body specifies an arbitrary structure X matching the signature S. Provided S denotes the semantic signature $(P)\mathcal{S}$, the denotation of B is determined in the context extended with the assumption $[\mathrm{X} : \mathcal{S}]$. The side condition on P ensures that the variables are treated as indeterminate types. These variables are subsequently discharged and become bound in the compound signature $(P \cup Q)\mathrm{X} : \mathcal{S}, \mathcal{S}'$.

Signature Expressions $\boxed{\mathcal{C} \vdash \mathrm{S} \triangleright \mathcal{L}}$

Signature expressions denote semantic signatures.

$$\frac{\mathcal{C} \vdash \mathrm{B} \triangleright \mathcal{L}}{\mathcal{C} \vdash \textbf{sig}\ \mathrm{B}\ \textbf{end} \triangleright \mathcal{L}} \tag{E-6}$$

Type Occurrences $\boxed{\mathcal{C} \vdash \mathrm{do} \triangleright d}$

Type occurrences denote definable types.

$$\frac{\mathrm{t} \in \mathrm{Dom}(\mathcal{C}) \quad \mathcal{C}(\mathrm{t}) = \tau}{\mathcal{C} \vdash \mathrm{t} \triangleright \hat{\eta}(\tau)} \tag{E-7}$$

$$\frac{\mathcal{C} \vdash \mathrm{sp} : \mathcal{S} \quad \mathrm{t} \in \mathrm{Dom}(\mathcal{S}) \quad \mathcal{S}(\mathrm{t}) = \tau}{\mathcal{C} \vdash \mathrm{sp.t} \triangleright \hat{\eta}(\tau)} \tag{E-8}$$

(E-7) We assume that the Core judgements expect type occurrences to denote definable types. However, type identifiers denote types, not definable types. By applying the operation $\hat{\eta}$ we convert the type τ denoted by t into the equivalent definable type $\hat{\eta}(\tau)$. A similar comment applies to Rule (E-8).

(E-8) Observe how the denotation of the type projection sp.t is determined statically by inspecting the type $\mathcal{S}$ of the structure path sp. In particular, we do not need to evaluate the term sp to determine the denotation of its type component.

Classification Rules

Structure Paths $\boxed{\mathcal{C} \vdash \mathrm{sp} : \mathcal{S}}$

Structure paths are classified by semantic structures.

$$\frac{\mathrm{X} \in \mathrm{Dom}(\mathcal{C}) \quad \mathcal{C}(\mathrm{X}) = \mathcal{S}}{\mathcal{C} \vdash \mathrm{X} : \mathcal{S}} \tag{E-9}$$

$$\frac{\mathcal{C} \vdash \mathrm{sp} : \mathcal{S}' \quad \mathrm{X} \in \mathrm{Dom}(\mathcal{S}') \quad \mathcal{S}'(\mathrm{X}) = \mathcal{S}}{\mathcal{C} \vdash \mathrm{sp.X} : \mathcal{S}} \tag{E-10}$$

Value Occurrences $\boxed{\mathcal{C} \vdash \mathrm{vo} : v}$

Value occurrences are classified by value types.

$$\frac{\mathrm{x} \in \mathrm{Dom}(\mathcal{C}) \quad \mathcal{C}(\mathrm{x}) = v}{\mathcal{C} \vdash \mathrm{x} : v} \tag{E-11}$$

$$\frac{\mathcal{C} \vdash \mathrm{sp} : \mathcal{S} \quad \mathrm{x} \in \mathrm{Dom}(\mathcal{S}) \quad \mathcal{S}(\mathrm{x}) = v}{\mathcal{C} \vdash \mathrm{sp.x} : v} \tag{E-12}$$

Generative Classification Rules

The generative classification judgements $\mathcal{C}, N \vdash \mathrm{b} : \mathcal{S} \Rightarrow M$ and $\mathcal{C}, N \vdash \mathrm{s} : \mathcal{S} \Rightarrow M$ are special. In each, the phrase is classified with respect to both a context $\mathcal{C}$ and a set of type variables N. N is a *state* capturing the set of variables generated so far. It should be a superset of $\mathrm{FV}(\mathcal{C})$. The phrase is related to a semantic structure $\mathcal{S}$ and a set of variables M. M is the set of "new" type variables generated during the classification of the phrase. Inspecting the rules, we can see that these variable sets are threaded through the classification tree in a global, state-like manner. Generated variables are accumulated in the state as classification traverses the structure of the phrase. The rules ensure that M is distinct from N. It is in this sense that the variables of M are new.

Structure Bodies $\boxed{\mathcal{C}, N \vdash \mathrm{b} : \mathcal{S} \Rightarrow M}$

Structure bodies are classified by semantic structures.

$$\frac{\mathcal{C} \vdash \mathrm{d} \triangleright d \quad \mathcal{C}[\mathrm{t} = d], N \vdash \mathrm{b} : \mathcal{S} \Rightarrow M \quad \mathrm{t} \notin \mathrm{Dom}(\mathcal{S})}{\mathcal{C}, N \vdash \textbf{type}\ \mathrm{t} = \mathrm{d}; \mathrm{b} : \mathrm{t} = d, \mathcal{S} \Rightarrow M} \tag{E-13}$$

$$\frac{\mathcal{C} \vdash \mathrm{e} : v \quad \mathcal{C}[\mathrm{x} : v], N \vdash \mathrm{b} : \mathcal{S} \Rightarrow M \quad \mathrm{x} \notin \mathrm{Dom}(\mathcal{S})}{\mathcal{C}, N \vdash \textbf{val}\ \mathrm{x} = \mathrm{e}; \mathrm{b} : \mathrm{x} : v, \mathcal{S} \Rightarrow M} \tag{E-14}$$

$$\frac{\mathcal{C}, N \vdash \mathrm{s} : \mathcal{S} \Rightarrow P \quad \mathcal{C}[\mathrm{X} : \mathcal{S}], N \cup P \vdash \mathrm{b} : \mathcal{S}' \Rightarrow Q \quad \mathrm{X} \notin \mathrm{Dom}(\mathcal{S}')}{\mathcal{C}, N \vdash \textbf{structure}\ \mathrm{X}\ =\ \mathrm{s;b} : \mathrm{X} : \mathcal{S}, \mathcal{S}' \Rightarrow P \cup Q} \tag{E-15}$$

$$\frac{\mathcal{C}, N \vdash \mathrm{s} : \mathcal{S}' \Rightarrow P \quad \mathcal{C}[\mathrm{X} : \mathcal{S}'], N \cup P \vdash \mathrm{b} : \mathcal{S} \Rightarrow Q}{\mathcal{C}, N \vdash \textbf{local}\ \mathrm{X}\ =\ \mathrm{s}\ \textbf{in}\ \mathrm{b} : \mathcal{S} \Rightarrow P \cup Q} \tag{E-16}$$

$$\frac{\begin{array}{ll}\mathcal{C} \vdash \mathrm{S} \triangleright (P)\mathcal{S}' & P \cap N = \emptyset \\ \mathcal{C}[\mathrm{X} : \mathcal{S}'], N \cup P \vdash \mathrm{s} : \mathcal{S}'' \Rightarrow Q & \mathcal{C}[\mathrm{F} : (P)(\mathcal{S}', (Q)\mathcal{S}'')], N \vdash \mathrm{b} : \mathcal{S} \Rightarrow M\end{array}}{\mathcal{C}, N \vdash \textbf{functor}\ \mathrm{F}\ (\mathrm{X} : \mathrm{S})\ =\ \mathrm{s}\ \textbf{in}\ \mathrm{b} : \mathcal{S} \Rightarrow M} \tag{E-17}$$

$$\frac{}{\mathcal{C}, N \vdash \epsilon_{\mathrm{b}} : \epsilon_{\mathcal{S}} \Rightarrow \emptyset} \tag{E-18}$$

(E-13) The structure body defines t as the denotation of d: the remainder of the body b is classified in the context extended with the assumption $[\mathrm{t} = d]$. The types generated by the entire phrase are just the types generated by classifying b. The component is added to the resulting structure.

(E-14) The structure body defines x as the value expression e. Provided e has value type v, the remainder of the body b is classified in the context extended with the assumption $[\mathrm{x} : v]$. The types generated by the entire phrase are just the type generated by classifying b. The type of x is recorded in the resulting structure.

(E-15) The structure body defines X as the structure expression s. Provided s has structure $\mathcal{S}$, generating new types P, the remainder of the body b is classified in the context extended by the typing assumption $[\mathrm{X} : \mathcal{S}]$, and the new state recording the additional types P. The set of types generated by the entire phrase is just the union of the sets returned by the classification of s and b. The type of X is recorded in the resulting structure.

(E-16) The rule is similar to Rule (E-15). The difference is that the definition of X is local b, and does not become a component of the resulting structure. Note that the type variables generated locally by s are still recorded in the output set.

(E-17) The functor argument's signature S denotes a semantic signature $(P)\mathcal{S}'$. The functor F should be applicable to any actual argument whose structure matches $(P)\mathcal{S}'$. To this end, the functor body s is classified in the context extended with the assumption $[\mathrm{X} : \mathcal{S}']$, and the state recording the additional types P. The side condition $P \cap N = \emptyset$ means that variables in P are treated as fresh parameters during the classification of s. Adding P to the state ensures that the new types in Q, generated by the body, are distinct from P. F is bound to the semantic functor $(P)(\mathcal{S}', (Q)\mathcal{S}'')$ before classifying the remaining definitions in b. We stress that the functor F is only defined locally for the classification of b: it does not become a component of the resulting structure.

Structure Expressions $\boxed{\mathcal{C}, N \vdash \mathrm{s} : \mathcal{S} \Rightarrow M}$

Structure expressions are classified by semantic structures.

$$\frac{\mathcal{C} \vdash \mathrm{sp} : \mathcal{S}}{\mathcal{C}, N \vdash \mathrm{sp} : \mathcal{S} \Rightarrow \emptyset} \tag{E-19}$$

$$\frac{\mathcal{C}, N \vdash \mathrm{b} : \mathcal{S} \Rightarrow M}{\mathcal{C}, N \vdash \textbf{struct}\ \mathrm{b}\ \textbf{end} : \mathcal{S} \Rightarrow M} \tag{E-20}$$

$$\frac{\begin{array}{l} \mathcal{C}, N \vdash \mathrm{s} : \mathcal{S}' \Rightarrow P \\ \mathcal{C}(\mathrm{F}) = (Q)(\mathcal{S}'', (Q')\mathcal{S}''') \\ \mathcal{S}' \succeq \varphi(\mathcal{S}'') \\ \mathrm{Dom}(\varphi) = Q \\ \varphi((Q')\mathcal{S}''') = (P')\mathcal{S} \\ P' \cap (N \cup P) = \emptyset \end{array}}{\mathcal{C}, N \vdash \mathrm{F}\ \mathrm{s} : \mathcal{S} \Rightarrow P \cup P'} \tag{E-21}$$

$$\frac{\begin{array}{ll} \mathcal{C}, N \vdash \mathrm{s} : \mathcal{S} \Rightarrow P & \mathcal{C} \vdash \mathrm{S} \rhd \Lambda P'.\mathcal{S}' \\ \mathcal{S} \succeq \varphi(\mathcal{S}') & \mathrm{Dom}(\varphi) = P' \end{array}}{\mathcal{C}, N \vdash \mathrm{s} \succeq \mathrm{S} : \varphi(\mathcal{S}') \Rightarrow P} \tag{E-22}$$

$$\frac{\begin{array}{ll} \mathcal{C}, N \vdash \mathrm{s} : \mathcal{S} \Rightarrow P & \\ \mathcal{C} \vdash \mathrm{S} \rhd \Lambda P'.\mathcal{S}' & N \cap P' = \emptyset \\ \mathcal{S} \succeq \varphi(\mathcal{S}') & \mathrm{Dom}(\varphi) = P' \end{array}}{\mathcal{C}, N \vdash \mathrm{s} \setminus \mathrm{S} : \mathcal{S}' \Rightarrow P'} \tag{E-23}$$

(E-19) Classifying a structure path does not generate any new types.

(E-21) To classify a functor application, we first classify the actual argument s to obtain its structure $\mathcal{S}'$, generating the new types P. The application is well-typed, provided there is some realisation φ of the functor's type parameters Q such that $\mathcal{S}'$ enriches the realised structure $\varphi(\mathcal{S}'')$. In other words, we require that $\mathcal{S}$ *matches* the functor's argument signature $(Q)\mathcal{S}''$. Moreover, the structure $\mathcal{S}$ of the application is obtained by applying the *same* realisation to the functor result $(Q')\mathcal{S}'''$, yielding the signature $(P')\mathcal{S}$. This application of φ propagates the realisation from the functor's actual argument to its result. Variables in P' are the new types generated by the functor: they must be chosen so that they are distinct from N and P. The application itself generates new types $P \cup P'$. Intuitively, this is the sum of the types obtained by evaluating both the actual argument and the body of the functor.

(E-22) The curtailment $\mathrm{s} \succeq \mathrm{S}$ can be classified provided: s has structure $\mathcal{S}$, generating new types P; S denotes some signature $(P')\mathcal{S}'$; and $\mathcal{S}$ matches this signature (via φ). Since we merely require that $\mathcal{S}$ enriches $\varphi(\mathcal{S}')$, the structure $\varphi(\mathcal{S}')$ resulting from the curtailment may have fewer and less general components than $\mathcal{S}$. However, by the definition of enrichment, those type components that remain will denote the *same* types as in $\mathcal{S}$. The phrase generates the types generated by s.

(E-23) The abstraction $\mathrm{s} \setminus \mathrm{S}$ can be classified provided: s has structure $\mathcal{S}$, generating new types P; S denotes some signature $(P')\mathcal{S}'$; and $\mathcal{S}$ matches this signature (via φ). Since we merely require that $\mathcal{S}$ enriches $\varphi(\mathcal{S}')$, the structure $\mathcal{S}'$ resulting from the abstraction may have fewer and less general components than $\mathcal{S}$. Moreover, since the result of the abstraction is $\mathcal{S}'$ and not its realisation $\varphi(\mathcal{S}')$, the actual realisation of types in P' is effectively forgotten. Intuitively, these types are made abstract by replacing them with the newly generated variables P'. Their freshness is expressed by the side condition $N \cap P' = \emptyset$.

This completes the definition of Modules.

3.2 An Example Core Language: Core-ML

In this section, we describe a particular Core language: Core-ML. We include the definition of Core-ML to provide a familiar, concrete example.

3.2.1 Phrase Classes

Definition 3.20 (Core-ML Phrase Classes).

(cf. Hypothesis 3.1.)

Core-ML is a typed language providing the phrase classes shown in Figure 3.10. The phrase classes SimTypVar, CoreId and SimTyp are specific

$'\mathrm{a} \in \mathrm{SimTypVar}$	simple type variables
$\mathrm{i} \in \mathrm{CoreId}$	λ-bound identifiers

(a) Identifiers

$\mathrm{k} \in \mathrm{DefKind}$	kinds
$\mathrm{u} \in \mathrm{SimTyp}$	simple types
$\mathrm{d} \in \mathrm{DefTyp}$	definable types
$\mathrm{v} \in \mathrm{ValTyp}$	value types

(b) Type Syntax

$\mathrm{e} \in \mathrm{ValExp}$	value

(c) Term Syntax

Figure 3.10: Core-ML Phrase Classes.

CoreId	$\stackrel{\text{def}}{=}$	$\{\mathbf{i}, \mathbf{j}, \ldots\}$	λ-bound identifiers
SimTypVar	$\stackrel{\text{def}}{=}$	$\{'\mathrm{a}, '\mathrm{b}, \ldots\}$	simple type variables
DefKind	$\stackrel{\text{def}}{=}$	$\{0, 1, 2, 3, \ldots\}$	kinds (arities)
u	$::=$	$'\mathrm{a}$	
	$\mid$	$\mathrm{u} \to \mathrm{u}'$	function space
	$\mid$	$\mathrm{do}(\mathrm{u}_0, \ldots, \mathrm{u}_{\mathrm{k}-1})$	*type occurrence*
d	$::=$	$\Lambda('\mathrm{a}_0, \ldots, '\mathrm{a}_{\mathrm{k}-1}).\mathrm{u}$	parameterised simple type
v	$::=$	$\forall'\mathrm{a}_0, \ldots, '\mathrm{a}_{n-1}.\mathrm{u}$	polymorphic simple type
e	$::=$	i	identifier
	$\mid$	$\lambda\mathrm{i.e}$	λ-abstraction
	$\mid$	$\mathrm{e}\ \mathrm{e}'$	application
	$\mid$	vo	*value occurrence*

Figure 3.11: Core-ML Grammar

$'a \in \mathit{SimTypVar}$	simple type variables
$u \in \mathit{SimTyp}$	simple types
$d^{\mathrm{k}} \in \mathit{DefTyp}^{\mathrm{k}}$	definable types
$v \in \mathit{ValTyp}$	value types
$C \in \mathit{CoreContext}$	Core contexts

Figure 3.12: Core-ML Semantic Objects

to Core-ML, the others are those required by Hypothesis 3.1. Figure 3.11 presents Core-ML's grammar.

Variables $'\mathrm{a} \in \mathrm{SimTypVar}$ range over simple types. Identifiers $\mathrm{i} \in \mathrm{CoreId}$ range over simply typed values and are introduced by Core-ML λ-abstractions.

The kind $\mathrm{k} \in \mathrm{DefKind}$ of a definable type is a natural number describing its *arity*, or the number of simple type arguments it expects.

Simple types $\mathrm{u} \in \mathrm{SimTyp}$ are constructed from simple type variables, the arrow type constructor and applications of type occurrences $\mathrm{do} \in \mathrm{TypOcc}$, of the Modules language, to k-tuples of simple type arguments.

Definable types $\mathrm{d} \in \mathrm{DefTyp}$ are parameterised simple types, mapping k-tuples of simple type arguments to simple types.

Value expressions are specified by value types $\mathrm{v} \in \mathrm{ValTyp}$. These are universally quantified simple types, conventionally called *type schemes*. The quantification over type variables expresses polymorphism.

Value expressions $\mathrm{e} \in \mathrm{ValExp}$ are constructed from identifiers, abstractions, function application, and value occurrences $\mathrm{vo} \in \mathrm{ValOcc}$ of the Modules language.

Notation. For $k = 0$ we will often write $\Lambda('\mathrm{a}_0, \ldots, '\mathrm{a}_{k-1}).\mathrm{u}$, $\forall '\mathrm{a}_0, \ldots, '\mathrm{a}_{k-1}.\mathrm{u}$ and $\mathrm{do}(\mathrm{u}_0, \ldots, \mathrm{u}_{k-1})$ in the abbreviated forms u, u and do, respectively, when no confusion can arise. We shall also omit the parentheses enclosing a 1-tuple of simple type parameters or simple type arguments.

3.2.2 Semantic Objects

Definition 3.21 (Core-ML Semantic Objects).

(cf. Hypothesis 3.2.)

The semantic objects of Core-ML are defined in Figures 3.12 and 3.13. The sets *SimTypVar* and *SimTyp* are specific to Core-ML, the others are those required by Hypothesis 3.2.

$d^{\mathrm{k}} \in \mathit{DefTyp}^{\mathrm{k}}$	$::=$	$\Lambda({}'a_0, \ldots, {}'a_{\mathrm{k}-1}).u$	parameterised simple type
		(provided ${}'a_0, \ldots, {}'a_{\mathrm{k}-1}$ distinct)	
$v \in \mathit{ValTyp}$	$::=$	$\forall {}'a_0, \ldots, {}'a_{n-1}.u$	polymorphic simple type
		(provided ${}'a_0, \ldots, {}'a_{n-1}$ distinct)	
$u \in \mathit{SimTyp}$	$::=$	${}'a$	simple type variable
	$\mid$	$u \rightarrow u'$	function space
	$\mid$	$\nu^{\mathrm{k}}(u_0, \ldots, u_{\mathrm{k}-1})$	type name occurrence

$$C \in \mathit{CoreContext} \overset{\text{def}}{=} \left\{ \mathcal{C}_{\mathrm{i}} \cup \mathcal{C}_{'\mathrm{a}} \;\middle|\; \begin{array}{l} \mathcal{C}_{\mathrm{i}} \in \mathrm{CoreId} \overset{\text{fin}}{\rightarrow} \mathit{SimTyp}, \\ \mathcal{C}_{'\mathrm{a}} \in \mathrm{SimTypVar} \overset{\text{fin}}{\rightarrow} \mathit{SimTyp} \end{array} \right\}$$

Figure 3.13: Semantic Objects of Core-ML

Semantic simple type variables ${}'a \in \mathit{SimTypVar}$ are the denotations of syntactic type variables ${}'\mathrm{a} \in \mathrm{SimTypVar}$. Definable types $d^{\mathrm{k}} \in \mathit{DefTyp}^{\mathrm{k}}$ are k-ary parameterised simple types. Value types $v \in \mathit{ValTyp}$ are universally quantified simple types. Semantic simple types $u \in \mathit{SimTyp}$ are the denotations of syntactic simple types $\mathrm{u} \in \mathrm{SimTyp}$. Notice that a semantic simple type can consist of an application of a type name $\nu^{\mathrm{k}} \in \mathit{TypNam}^{\mathrm{k}}$, of the Modules language, to a k-tuple of simple type arguments.

Definable types $\Lambda({}'a_0, \ldots, {}'a_{\mathrm{k}-1}).u$ and value types $\forall {}'a_0, \ldots, {}'a_{n-1}.u$ are binding constructs. We identify definable types and value types that are equivalent up to renamings of bound simple type variables.

Remark 3.2.1 *(Relating Type Phrases to their Denotations).* Observe that the structure of corresponding syntactic and semantic objects is almost isomorphic. The essential difference between them is this: while simple type phrases may contain applications of type occurrence phrases, semantic simple types may not; instead, they allow for (well-kinded) applications of type names. It should come as no surprise that the denotation judgements merely replace type occurrences by expanding their denotations, preserving the structure of the original phrase in all other respects.

Notation. For $k = 0$ we will often write $\Lambda({}'a_0, \ldots, {}'a_{k-1}).u$, $\forall {}'a_0, \ldots, {}'a_{k-1}.u$ and $\nu(u_0, \ldots, u_{k-1})$ in the abbreviated forms u, u and ν respectively, when no confusion can arise. We shall also omit the parentheses enclosing a 1-tuple of simple type parameters or simple type arguments.

Definition 3.22 (Substitutions). A substitution:

$$\sigma \text{ or } \{'a_0 \mapsto u_0, \ldots, 'a_{k-1} \mapsto u_{k-1}\} \in \mathit{Subst} = \mathit{SimTypVar} \overset{\text{fin}}{\to} \mathit{SimTyp}$$

is a finite map mapping simple type variables to simple types.

Definition 3.23 (Core Contexts). Core contexts $C \in \mathit{CoreContext}$ map λ-bound identifiers to semantic simple types, and (syntactic) simple type variables to (semantic) simple types.

The following operations extend contexts with Core assumptions.

$$\begin{array}{rcl} \_[\_ = \_] & \in & (\mathit{Context} \times \mathrm{SimTypVar} \times \mathit{SimTyp}) \to \mathit{Context} \\ \mathcal{C}['\mathrm{a} = u] & \overset{\text{def}}{=} & \mathcal{C} + \{'\mathrm{a} \mapsto u\} \end{array}$$

$$\begin{array}{rcl} \_[\_ : \_] & \in & (\mathit{Context} \times \mathrm{CoreId} \times \mathit{SimTyp}) \to \mathit{Context} \\ \mathcal{C}[\mathrm{i} : u] & \overset{\text{def}}{=} & \mathcal{C} + \{\mathrm{i} \mapsto u\} \end{array}$$

Definition 3.24 (Free and Bound Simple Type Variables). For any semantic object $\mathcal{O}$, we let $\mathrm{FTVS}(\mathcal{O})$ denote the set of simple type variables occurring *free* in $\mathcal{O}$, where the notions of free and bound are defined as usual.

Definition 3.25 (η-expansion of Type Names).
(cf. Hypothesis 3.5.)
We define the kind-preserving operation η as follows:

$$\begin{array}{rcl} \eta(\_) & \in & \cup_{\mathrm{k} \in \mathrm{DefKind}} \mathit{TypNam}^{\mathrm{k}} \to \mathit{DefTyp}^{\mathrm{k}} \\ \eta(\nu^{\mathrm{k}}) & \overset{\text{def}}{=} & \Lambda('a_0, \ldots, 'a_{\mathrm{k}-1}).\nu^{\mathrm{k}}('a_0, \ldots, 'a_{\mathrm{k}-1}) \\ & & (\text{provided } \forall i \in [\mathrm{k}].'a_i \notin \mathrm{FTVS}(\nu^{\mathrm{k}}) \cup \{'a_0, \ldots, 'a_{i-1}\}) \end{array}$$

The proviso that $'a_0, \ldots, 'a_{\mathrm{k}-1}$ are distinct and fresh[4] ensures that the function is injective.

The operation allows any type name ν to be viewed as an equivalent definable type $\eta(\nu)$.

Definition 3.26 (Realisation of Definable and Value Types).
(cf. Hyp. 3.14.)

[4]In this chapter, a type name ν^{k} can never contain free simple type variables. However, when we generalise type names in Chapter 5, the condition $'a_i \notin \mathrm{FTVS}(\nu^{\mathrm{k}})$ will no longer be vacuous.

Recall that realisations were defined in Definition 3.13. We now define the effect of applying a realisation to a Core semantic object as follows:

$$\begin{array}{rcl} \_\,(\_) & \in & (\mathit{Real} \times \mathit{DefTyp}^{\mathrm{k}}) \to \mathit{DefTyp}^{\mathrm{k}} \quad \text{(for each k} \in \text{DefKind)} \\ \varphi\left(d^{\mathrm{k}}\right) & \stackrel{\text{def}}{=} & \begin{cases} \Lambda({}'a_0,\ldots,{}'a_{\mathrm{k}-1}).\varphi\,(u) & \text{provided} \\ & d^{\mathrm{k}} = \Lambda({}'a_0,\ldots,{}'a_{\mathrm{k}-1}).u \\ & \text{and } \forall i \in [\mathrm{k}].{}'a_i \notin \mathrm{FTVS}(\varphi) \end{cases} \end{array}$$

$$\begin{array}{rcl} \_\,(\_) & \in & (\mathit{Real} \times \mathit{ValTyp}) \to \mathit{ValTyp} \\ \varphi\,(v) & \stackrel{\text{def}}{=} & \begin{cases} \forall{}'a_0,\ldots,{}'a_{n-1}.\varphi\,(u) & \text{provided} \\ & v = \forall{}'a_0,\ldots,{}'a_{n-1}.u \\ & \text{and } \forall i \in [n].{}'a_i \notin \mathrm{FTVS}(\varphi) \end{cases} \end{array}$$

$$\begin{array}{rcl} \_\,(\_) & \in & (\mathit{Real} \times \mathit{SimTyp}) \to \mathit{SimTyp} \\ \varphi\,({}'a) & \stackrel{\text{def}}{=} & {}'a \\ \varphi\,(u \to u') & \stackrel{\text{def}}{=} & \varphi\,(u) \to \varphi\,(u') \\ \varphi\,(\nu(u_0,\ldots,u_{\mathrm{k}-1})) & \stackrel{\text{def}}{=} & \begin{cases} \nu'(\varphi\,(u_0)\,,\ldots,\varphi\,(u_{\mathrm{k}-1})) & \text{if } \varphi\,(\nu) = \nu' \\ \{{}'a_i \mapsto \varphi\,(u_i)\,|i \in [\mathrm{k}]\}\,(u) & \text{if } \varphi(\nu) = \\ & \Lambda({}'a_0,\ldots,{}'a_{\mathrm{k}-1}).u \end{cases} \end{array}$$

Note the reduction step in the last case defining $\varphi\,(\nu(u_0,\ldots,u_{\mathrm{k}-1}))$. This case occurs whenever the type name ν is realised by a definable type. We need to perform the reduction in order to respect the definition of simple types that only allows applications of type names, not definable types, to simple type arguments. Intuitively, simple types are kept in normal form.

We will need a relation that relates value types to their simple type instances (this relation is specific to Core-ML):

Definition 3.27 (Value Types Generalising Simple Types). We define the relation:

$$\_ \succ \_ \in \mathit{ValTyp} \times \mathit{SimTyp}$$

as follows.

A value type $v \equiv \forall{}'a_0,\ldots,{}'a_{n-1}.u$ *generalises* a simple type u', written $v \succ u'$ if, and only if, there is a substitution σ with $\mathrm{Dom}(\sigma) = \{{}'a_0,\ldots,{}'a_{n-1}\}$ such that $\sigma(u) = u'$.

The generalisation relation is the basis for defining enrichment on value types.

$\mathcal{C} \vdash \mathrm{u} \triangleright u$

In context $\mathcal{C}$, simple type u denotes simple type u.

$\mathcal{C} \vdash \mathrm{d} \triangleright d$

In context $\mathcal{C}$, definable type d denotes definable type d.

$\mathcal{C} \vdash \mathrm{v} \triangleright v$

In context $\mathcal{C}$, value type v denotes value type v.

(a) Denotation Judgements

$\mathcal{C} \vdash \mathrm{e} : u$

In context $\mathcal{C}$, value expression e has simple type u.

$\mathcal{C} \vdash \mathrm{e} : v$

In context $\mathcal{C}$, value expression e has value type v.

(b) Classification Judgements

Figure 3.14: Core Judgements.

Definition 3.28 (Enrichment between Value Types).

(cf. Hypothesis 3.16.)

We define the relation:

$$\_ \succeq \_ \in \mathit{ValTyp} \times \mathit{ValTyp}$$

as follows.

A value type v *enriches* another value type v', written $v \succeq v'$, if and only if, for every simple type u, $v \succ u$ whenever $v' \succ u$.

It is well-known that $\_ \succeq \_$ is a pre-order [Mil78]. It is easy to show that it is closed under realisation.

3.2.3 Static Semantics

Definition 3.29 (Core-ML Judgements).

(cf. Hypothesis 3.19.)

The denotation and classification judgements of Core-ML are presented in Figure 3.14.

The judgements $\mathcal{C} \vdash \mathrm{u} \triangleright u$ and $\mathcal{C} \vdash \mathrm{e} : u$ are specific to Core-ML (the others are those required by Hypothesis 3.19). The former, $\mathcal{C} \vdash \mathrm{u} \triangleright u$, relates a simple type expression to its denotation. The latter, $\mathcal{C} \vdash \mathrm{e} : u$, classifies a value expression by a simple type. In Core-ML, a value expression may inhabit more than one simple type. This judgement should be clearly distinguished from the judgement $\mathcal{C} \vdash \mathrm{e} : v$ (required by Hypothesis 3.19) that classifies an expression by a value type, i.e. a universally quantified simple type. As for ordinary ML [Mil78, Dam85], it is a property of Core-ML that every typable value expression has a *principal*, or most general, value type.

The judgements are defined by the following rules:

Denotation Rules

Simple Types $\boxed{\mathcal{C} \vdash \mathrm{u} \rhd u}$

Simple types denote semantic simple types.

$$\frac{\mathcal{C}(\mathrm{'a}) = u}{\mathcal{C} \vdash \mathrm{'a} \rhd u} \tag{C-1}$$

$$\frac{\mathcal{C} \vdash \mathrm{u} \rhd u \quad \mathcal{C} \vdash \mathrm{u}' \rhd u'}{\mathcal{C} \vdash \mathrm{u} \to \mathrm{u}' \rhd u \to u'} \tag{C-2}$$

$$\frac{\mathcal{C} \vdash \mathrm{do} \rhd \Lambda('a_0, \ldots, 'a_{\mathrm{k}-1}).u \quad \forall i \in [\mathrm{k}]. \quad \mathcal{C} \vdash \mathrm{u}_i \rhd u_i}{\mathcal{C} \vdash \mathrm{do}(\mathrm{u}_0, \ldots, \mathrm{u}_{\mathrm{k}-1}) \rhd \{'a_i \mapsto u_i | i \in [\mathrm{k}]\}\,(u)} \tag{C-3}$$

(C-1) A simple type variable denotes only if it is bound in the context.

(C-3) The denotation of a type occurrence is immediately applied to its arguments: simple types are kept in normal form. The application denotes only if the arity of the definable type matches the number of its actual arguments. Note that the first premise is an instance of the Modules judgement $\mathcal{C} \vdash \mathrm{do} \rhd d$.

Definable Types $\boxed{\mathcal{C} \vdash \mathrm{d} \rhd d}$

Definable types denote semantic definable types.

$$\frac{\begin{array}{c}\mathcal{C}[\mathrm{'a}_0 = 'a_0] \cdots [\mathrm{'a}_{\mathrm{k}-1} = 'a_{\mathrm{k}-1}] \vdash \mathrm{u} \rhd u \\ \forall i \in [\mathrm{k}].'a_i \notin \mathrm{FTVS}(\mathcal{C}) \cup \{'a_0, \ldots, 'a_{i-1}\}\end{array}}{\mathcal{C} \vdash \Lambda(\mathrm{'a}_0, \cdots, \mathrm{'a}_{\mathrm{k}-1}).\mathrm{u} \rhd \Lambda('a_0, \cdots, 'a_{\mathrm{k}-1}).u} \tag{C-4}$$

(C-4) Syntactic simple type variables are bound to distinct and fresh semantic type variables before determining the denotation of u.

Value Types $\boxed{\mathcal{C} \vdash \mathrm{v} \triangleright v}$

Value types denote semantic value types.

$$\frac{\begin{array}{l}\mathcal{C}[{}'\mathrm{a}_0 = {}'a_0]\cdots[{}'\mathrm{a}_{n-1} = {}'a_{n-1}] \vdash \mathrm{u} \triangleright u \\ \forall i \in [n].{}'a_i \notin \mathrm{FTVS}(\mathcal{C}) \cup \{{}'a_0, \ldots, {}'a_{i-1}\}\end{array}}{\mathcal{C} \vdash \forall{}'\mathrm{a}_0, \cdots, {}'\mathrm{a}_{n-1}.\mathrm{u} \triangleright \forall{}'a_0, \cdots, {}'a_{n-1}.u} \quad \text{(C-5)}$$

(C-5) See the comment to Rule (C-4).

Classification Rules

Monomorphic Values $\boxed{\mathcal{C} \vdash \mathrm{e} : u}$

In the first instance, value expressions are classified by semantic simple types.

$$\frac{\mathcal{C}(\mathrm{i}) = u}{\mathcal{C} \vdash \mathrm{i} : u} \quad \text{(C-6)}$$

$$\frac{\mathcal{C}[\mathrm{i} : u] \vdash \mathrm{e} : u'}{\mathcal{C} \vdash \lambda\mathrm{i}.\mathrm{e} : u \to u'} \quad \text{(C-7)}$$

$$\frac{\mathcal{C} \vdash \mathrm{e} : u' \to u \quad \mathcal{C} \vdash \mathrm{e}' : u'}{\mathcal{C} \vdash \mathrm{e}\ \mathrm{e}' : u} \quad \text{(C-8)}$$

$$\frac{\mathcal{C} \vdash \mathrm{vo} : v \quad v \succ u}{\mathcal{C} \vdash \mathrm{vo} : u} \quad \text{(C-9)}$$

(C-6) Identifiers introduced by λ-abstractions have simple types. See rule (C-7).

(C-7) In Core-ML, to ensure decidable type inference, identifiers bound by λ-abstractions range over monomorphic value expressions classified by simple types.

(C-9) A value occurrence vo has any simple type that is an instance of its polymorphic value type. Note that the first premise is an instance of the Modules judgement $\mathcal{C} \vdash \mathrm{vo} : v$.

Polymorphic Values $\boxed{\mathcal{C} \vdash \mathrm{e} : v}$

The definition of Modules assumes that value expression are classified by value types. For Core-ML, a principal value type for an expression can be specified in terms of its possible simple types.

$$\frac{\begin{array}{c}\mathcal{C} \vdash \mathrm{e} : u \quad \{'a_0, \ldots, 'a_{n-1}\} = \mathrm{FTVS}(u) \setminus \mathrm{FTVS}(\mathcal{C}) \\ \forall u'. \quad \mathcal{C} \vdash \mathrm{e} : u' \supset \forall 'a_0, \ldots, 'a_{n-1}.u \succ u'\end{array}}{\mathcal{C} \vdash \mathrm{e} : \forall 'a_0, \ldots, 'a_{n-1}.u} \tag{C-10}$$

(C-10) Informally, the rule states that e has value type v provided e has v's *generic* instance as a type, and every other simple type of e is an instance of v. The second condition is expressed by an infinitary premise. The rule specifies that v is a principal value type for e. (The idea of using an infinitary rule to force principality is similar to the use of "higher-order inference rules" in Kahrs, Sannella and Tarlecki's definition of Extended ML [KST97]).

This completes the definition of Core-ML.

3.3 Discussion of Mini-SML

We define Mini-SML as the language obtained by combining the definitions of Modules and Core-ML. Without being too pedantic, we define:

Definition 3.30 (Mini-SML). Mini-SML is obtained by:

- Defining the syntactic phrase classes of Mini-SML (Figures 3.2 and 3.10) as the least solutions to the grammar rules in Figures 3.3 and 3.11.
- Defining the semantic objects of Mini-SML (Figures 3.5 and 3.12) as the least solutions to the defining equations in Figures 3.6 and 3.13.
- Defining the judgements of Mini-SML (Figures 3.9 and 3.14) as the least relations closed under the rules in Sections 3.1.3 and 3.2.3.

Although Core-ML is much simpler than Core Standard ML, Mini-SML does model the prominent features of Standard ML Modules. There is one apparent omission that we should comment on.

$$
\begin{array}{l}
\mathbf{datatype}\ \mathrm{t}\ =\ \Lambda('\mathrm{a}_1,\ldots,'\mathrm{a}_\mathrm{k}).\mathrm{u}\ \mathbf{with}\ \mathrm{x},\mathrm{x}';\ \mathrm{b} \\
\overset{\mathrm{def}}{=}\ \mathbf{local\ structure}\ \mathrm{X} = \\
\qquad\qquad\qquad \begin{array}{ll}
\mathbf{struct} & \mathbf{type}\ \mathrm{t} = \Lambda('\mathrm{a}_1,\ldots,'\mathrm{a}_\mathrm{k}).\mathrm{u}; \\
 & \mathbf{val}\ \mathrm{x} = \lambda \mathrm{i}.\mathrm{i}; \\
 & \mathbf{val}\ \mathrm{x}' = \lambda \mathrm{i}.\mathrm{i} \\
\mathbf{end} \setminus & \\
\mathbf{sig} & \mathbf{type}\ \mathrm{t} : \mathrm{k}; \\
 & \mathbf{val}\ \mathrm{x} : \forall '\mathrm{a}_1,\ldots,'\mathrm{a}_\mathrm{k}.\mathrm{u} \to \mathrm{t}('\mathrm{a}_1,\ldots,'\mathrm{a}_\mathrm{k}); \\
 & \mathbf{val}\ \mathrm{x}' : \forall '\mathrm{a}_1,\ldots,'\mathrm{a}_\mathrm{k}.\mathrm{t}('\mathrm{a}_1,\ldots,'\mathrm{a}_\mathrm{k}) \to \mathrm{u} \\
\mathbf{end} &
\end{array} \\
\quad \begin{array}{ll}
\mathbf{in} & \mathbf{type}\ \mathrm{t} = \Lambda('\mathrm{a}_1,\ldots,'\mathrm{a}_\mathrm{k}).\mathrm{X}.\mathrm{t}('\mathrm{a}_1,\ldots,'\mathrm{a}_\mathrm{k}); \\
 & \mathbf{val}\ \mathrm{x} = \mathrm{X}.\mathrm{x}; \\
 & \mathbf{val}\ \mathrm{x}' = \mathrm{X}.\mathrm{x}'; \\
 & \mathrm{b} \\
\mathbf{end} &
\end{array}
\end{array}
$$

Figure 3.15: The datatype definition as an abbreviated structure body.

Remark 3.3.1 *(Standard ML's Datatype Phrases).* Readers familiar with Standard ML may be disconcerted by the omission of **datatype** definitions and specifications. In Standard ML, the phrase:

$$\mathbf{datatype}\ ('\mathrm{a}_1,\ldots,'\mathrm{a}_\mathrm{k})\mathrm{t} = C_1\ \mathbf{of}\ u_1|\cdots|C_n\ \mathbf{of}\ u_n;\ \mathrm{b}\quad (n \geq 1)$$

is used to define a "new" (possibly recursive) type t along with constructors C_i mediating between it and its representation. The relevant point here is that t is abstract *as soon as it is defined*, yet our description of Modules only supports abstraction *after encapsulation*, by applying a signature to a structure expression.

Adapting the notion to Core ML (which, for simplicity, has neither recursive types nor pattern-matching on constructors), we might at least expect an analogous phrase:

$$\mathbf{datatype}\ \mathrm{t}\ =\ \Lambda('\mathrm{a}_1,\ldots,'\mathrm{a}_\mathrm{k}).\mathrm{u}\ \mathbf{with}\ \ \mathrm{x},\mathrm{x}';\ \mathrm{b}$$

to introduce t as a new type with "constructor":

$$\mathrm{x} : \forall '\mathrm{a}_1,\ldots,'\mathrm{a}_\mathrm{k}.\mathrm{u} \to \mathrm{t}('\mathrm{a}_1,\ldots,'\mathrm{a}_\mathrm{k})$$

and "destructor":

$$\mathrm{x}' : \forall '\mathrm{a}_1,\ldots,'\mathrm{a}_\mathrm{k}.\mathrm{t}('\mathrm{a}_1,\ldots,'\mathrm{a}_\mathrm{k}) \to \mathrm{u}.$$

$$\begin{array}{l}
\textbf{datatype}\ \mathrm{t}\ =\ \Lambda('\mathrm{a}_1,\ldots,'\mathrm{a}_\mathrm{k}).\mathrm{u}\ \textbf{with}\ \mathrm{x},\mathrm{x}';\ \mathrm{B}\\
\stackrel{\mathrm{def}}{=}\ \textbf{type}\ \mathrm{t} : \mathrm{k};\\
\quad \textbf{val}\ \mathrm{x} : \forall'\mathrm{a}_1,\ldots,'\mathrm{a}_\mathrm{k}.\mathrm{u} \to \mathrm{t}('\mathrm{a}_1,\ldots,'\mathrm{a}_\mathrm{k});\\
\quad \textbf{val}\ \mathrm{x}' : \forall'\mathrm{a}_1,\ldots,'\mathrm{a}_\mathrm{k}.\mathrm{t}('\mathrm{a}_1,\ldots,'\mathrm{a}_\mathrm{k}) \to \mathrm{u};\\
\quad \mathrm{B}
\end{array}$$

Figure 3.16: The datatype specification as an abbreviated signature body.

Fortunately, this can be defined as an abbreviation for the structure body in Figure 3.15, where the identifier X must by chosen to be fresh with respect to the current context and the identifiers defined in b.

Similarly, we can define the datatype *specification*

$$\textbf{datatype}\ \mathrm{t}\ =\ \Lambda('\mathrm{a}_1,\ldots,'\mathrm{a}_\mathrm{k}).\mathrm{u}\ \textbf{with}\ \mathrm{x},\mathrm{x}';\ \mathrm{B}$$

as an abbreviation for the signature body in Figure 3.16.

Treating datatype phrases as syntactic sugar means that we can avoid polluting the generic Modules language and its semantics with constructs that are incidental to Core-ML.

It is easy to verify that these abbreviations give rise to the following derived rules:

$$\frac{\begin{array}{l}
\mathcal{C} \vdash \mathrm{d} \rhd \Lambda('a_1,\ldots,'a_\mathrm{k}).u\\
\alpha^\mathrm{k} \notin N\\
v \equiv \forall' a_1,\ldots,'a_\mathrm{k}.u \to \alpha^\mathrm{k}('a_1,\ldots,'a_\mathrm{k})\\
v' \equiv \forall' a_1,\ldots,'a_\mathrm{k}.\alpha^\mathrm{k}('a_1,\ldots,'a_\mathrm{k}) \to u\\
\mathcal{C}[\mathrm{t} = \alpha^\mathrm{k}][\mathrm{x} : v][\mathrm{x}' : v'], N \cup \{\alpha^\mathrm{k}\} \vdash \mathrm{b} : \mathcal{S}' \Rightarrow M\\
\mathrm{x} \neq \mathrm{x}' \quad \{\mathrm{t},\mathrm{x},\mathrm{x}'\} \cap \mathrm{Dom}(\mathcal{S}') = \emptyset \quad \mathcal{S} \equiv \mathrm{t} = \alpha^\mathrm{k}, \mathrm{x} : v, \mathrm{x}' : v', \mathcal{S}'
\end{array}}{\mathcal{C}, N \vdash \textbf{datatype}\ \mathrm{t}\ =\ \mathrm{d}\ \textbf{with}\ \mathrm{x},\mathrm{x}';\ \mathrm{b} : \mathcal{S} \Rightarrow \{\alpha^\mathrm{k}\} \cup M} \qquad \text{(C-11)}$$

$$\frac{\begin{array}{l}
\mathcal{C} \vdash \mathrm{d} \rhd \Lambda('a_1,\ldots,'a_\mathrm{k}).u\\
\alpha^\mathrm{k} \notin \mathrm{FV}(\mathcal{C})\\
v \equiv \forall' a_1,\ldots,'a_\mathrm{k}.u \to \alpha^\mathrm{k}('a_1,\ldots,'a_\mathrm{k})\\
v' \equiv \forall' a_1,\ldots,'a_\mathrm{k}.\alpha^\mathrm{k}('a_1,\ldots,'a_\mathrm{k}) \to u\\
\mathcal{C}[\mathrm{t} = \alpha^\mathrm{k}][\mathrm{x} : v][\mathrm{x}' : v'] \vdash \mathrm{B} \rhd (P)\mathcal{S}'\\
P \cap (\{\alpha^\mathrm{k}\} \cup \mathrm{FV}(v) \cup \mathrm{FV}(v')) = \emptyset\\
\mathrm{x} \neq \mathrm{x}' \quad \{\mathrm{t},\mathrm{x},\mathrm{x}'\} \cap \mathrm{Dom}(\mathcal{S}') = \emptyset \quad \mathcal{S} \equiv \mathrm{t} = \alpha^\mathrm{k}, \mathrm{x} : v, \mathrm{x}' : v', \mathcal{S}'
\end{array}}{\mathcal{C} \vdash \textbf{datatype}\ \mathrm{t}\ =\ \mathrm{d}\ \textbf{with}\ \mathrm{x},\mathrm{x}';\ \mathrm{B} \rhd (\{\alpha^\mathrm{k}\} \cup P)\mathcal{S}} \qquad \text{(C-12)}$$

These rules should be familiar to those acquainted with the static semantics of Standard ML [MTH90, MTH96].

3.3.1 Discussion of the Syntax

Like many other statically typed programming languages, the syntax of Mini-SML describes a language of types and terms. In Figures 3.2 and 3.10 we grouped phrase classes according to whether they belong to the type or term syntax of the language. However, a fundamental design principle of Modules is that related type and term definitions should be packaged together in syntactic units as structures. Access to these components is via the dot notation. As a result, the syntax of Mini-SML exhibits a curious feature that is atypical of most statically typed languages: terms may appear in types. The ultimate source of this dependency is the type projection sp.t. The phrase belongs to the syntax of types, but sp is a structure path, that, since it refers to a collection of both types and terms, belongs to the proper syntax of terms. At first sight, this raises the uncomfortable question of whether we need to *evaluate* the term sp in order to determine the type denoted by sp.t.

This merits further explanation. In typical, statically-typed programming languages, types and terms are kept distinct in the sense that types cannot contain occurrences of terms. For this simple reason alone, the equivalence of types, used in determining the classification of terms, can usually be decided without resorting to reasoning about the equivalence of terms (evaluation). For instance, the type theories we presented in Sections 2.2.1, 2.2.2, 2.2.3, and 2.2.4 were all of this nature.

In languages with dependent types, such as the type theories of Sections 2.2.5 and 2.2.6, this stratification is relaxed: not only can types contain terms, but more importantly, the notion of type equivalence actually depends on the notion of term equivalence. This dependency is manifest in the rules defining type equivalence, that refer, at least indirectly, to the judgement relating equivalent terms. Similarly, in a programming language with true dependent types, we would expect to see a dependency of the static semantics of the language on the dynamic semantics of the language, preventing the language from having a phase distinction between compile-time type-checking and run-time evaluation.

In Mini-SML, the fact that type phrases can contain term phrases is suggestive of a dependently-typed language. It is this syntactic idiosyncrasy, shared by Standard ML, that has inspired much of the research on the type structure of Standard ML to resort to the use of dependent types

[Mac86, HM93, HMM90]. Indeed, the use of dependent types lingers on in the more recent type-theoretic alternatives to Standard ML proposed in [HL94, Ler94, Ler96b, Ler95, Lil97, SH96, HS97]. By contrast, in the following sections and the next chapter, we shall present evidence to suggest that dependent types play no discernible role in the semantics of Mini-SML. Since Mini-SML is merely a cut down version of Standard ML, we will conclude with the counter-claim that dependent types have no role in the semantics of Standard ML.

3.3.2 Discussion of the Semantic Objects

As in Standard ML, the definition of Mini-SML distinguishes between syntactic type phrases and the semantic objects they denote. Moreover, it is the semantic objects of Mini-SML (Figures 3.4, 3.6 and 3.13), not the syntactic type phrases, that serve the role of types in the judgements defining the static semantics. To see this, observe that contexts relate identifiers to semantic objects and that the judgements of Mini-SML classify term phrases by semantic objects. The type phrases of Mini-SML do not play a direct role in the classification judgements. Instead, the classification judgements exploit the *denotations* of type phrases, where the denotation of a type phrase is obtained by a judgement that essentially translates the phrase to its meaning as a semantic object.

The fundamental distinction between type phrases and semantic objects is the following: while type phrases can contain occurrences of terms, semantic objects cannot. It is easy to see this because the definition of semantic objects is independent of the definition of terms. We noted that the dependency of syntactic types on terms is introduced by type occurrence phrases do $\in$ TypOcc, that allow type projections sp.t from terms sp $\in$ StrPath. Note that structure paths are phrases that evaluate to structures and are, in a sense, *first-order* objects like the term phrases of the simply typed λ-calculus of Section 2.2.1. Contrast this with the semantic counterpart of TypOcc, the set of semantic definable types *DefTyp*. Observe that a semantic definable cannot contain occurrences of terms, but may, instead, contain occurrences of type variables $\alpha \in \mathit{TypVar}$. Type variables range over semantic types, and for this reason, are *second-order* variables, just like the type variables we encountered in the simply typed λ-calculus of Section 2.2.1.

To a type theorist, Mini-SML's distinction between syntactic types and their semantic counterparts is odd: in most type theories, type phrases are interpreted syntactically, with more complicated type theories resorting to an equational theory on type phrases. Adopting this approach in Mini-SML, whose type phrases contain occurrences of first-order terms, would mean introducing some form of dependency of type equivalence on term equivalence. By imposing an additional layer of interpretation on type phrases, which reduces type phrases to semantic objects that do not contain first-order terms, Mini-SML manages to avoid this form of dependency. The denotation judgements of Mini-SML thus serve to eliminate first-order dependencies of type phrases on terms, replacing them with second-order dependencies of semantic objects on types. The reason Mini-SML distinguishes between syntactic and semantic types is a pragmatic one: it maintains the facade of a language with first-order dependent types, convenient for structuring programs. The facade masks an underlying, purely second-order type theory, formulated in terms of semantic objects.

3.3.3 Discussion of the Judgements

It should be clear that Mini-SML is a statically typed language. None of the rules of the static semantics refer to term equivalence judgements of the dynamic semantics of Mini-SML. In fact, we have not even defined these judgements.

When presenting the judgements of Mini-SML, we divided them into three separate, functionally related, groups of judgements: denotation, classification and generative classification judgements. Examining the roles of the judgments will help us understand the semantics.

Denotation Judgements

The denotation judgements all have the form $\mathcal{C} \vdash \mathrm{p} \triangleright p$. In each, the phrase p is a phrase from the syntax of types (not terms). Its denotation is determined with respect to a context of typing assumptions $\mathcal{C}$. The denotation p is an object in the semantic counterpart of the phrase class of p. The denotation judgements are used in the semantics to systematically replace type phrases by their denotations.

The key to understanding the purpose of the denotation judgements lies in understanding Rule (E-8):

$$\frac{\mathcal{C} \vdash \mathrm{sp} : \mathcal{S} \quad \mathrm{t} \in \mathrm{Dom}(\mathcal{S}) \quad \mathcal{S}(\mathrm{t}) = \tau}{\mathcal{C} \vdash \mathrm{sp.t} \triangleright \hat{\eta}(\tau)}$$

The rule states that the denotation $\hat{\eta}(\tau)$ of the "dependent" phrase sp.t is determined *statically* by inspecting the type and not the (run-time) value of sp. By systematically applying rules (E-7) and (E-8) the denotation judgements in Figure 3.9(a) serve to replace occurrences of syntactic term-dependencies by their non-dependent denotations. The denotation judgements translate the term-dependent type syntax of Mini-SML into the underlying, term-independent language of semantic objects.

Classification Judgements

The classification judgements all have the form $\mathcal{C} \vdash \mathrm{p} \ : \ o$. In each, the phrase p is a term phrase that is related to its classification o in the context $\mathcal{C}$. The judgement states that the term p has type o. We stress that o is a semantic object, not a syntactic type phrase. The classification judgements are clearly of a different nature from the denotation judgements discussed in the previous section: while a denotation judgement merely translates a syntactic type phrase to its semantic representation, a classification judgement relates a term to its type. Indeed, the denotation of a term phrase would be defined by the dynamic semantics of terms, which we have not presented. Classification judgements are thus instances of the familiar typing relations we encountered in our introduction to Type Theory (Section 2.2).

Generative Classification Judgements

The generative classification judgements also serve to classify terms of the Modules language by their types. The judgement $\mathcal{C}, N \vdash \mathrm{b} : \mathcal{S} \Rightarrow P$ relates the structure body b to its type $\mathcal{S}$ in the context $\mathcal{C}$. The judgement $\mathcal{C}, N \vdash \mathrm{s} : \mathcal{S} \Rightarrow P$ relates the structure expression s to its type $\mathcal{S}$ in the context $\mathcal{C}$. These judgements differ from the other classification judgements in that the classification of functor applications and abstractions appearing within b and s can generate "new" types. For this reason, the generative judgements take two additional arguments, the sets of type variables N and P. Both sets are finite. N records the state of type variables "generated" prior to the classification of the phrase. For soundness reasons, it should be a superset of the type variables occurring free in $\mathcal{C}$. Classification produces, besides the semantic object $\mathcal{S}$, the set of type variables P generated *during* the classification of the phrase. These new type variables may occur in $\mathcal{S}$. The rules defining the judgements enforce the condition that each variable in P is distinct from those variables in the current state N, and, by implication, from any types appearing in $\mathcal{C}$. It is in this sense that P captures the set of "new" types generated by the phrase. This invariant is maintained by threading the state through the derivation tree, updating it as new variables are generated by subphrases, functor applications and abstractions. This generative behaviour is captured by the following property:

Property 3.31 (Generativity).

- *If* $\mathcal{C}, N \vdash \mathrm{b} : \mathcal{S} \Rightarrow P$ *then* $N \cap P = \emptyset$*.*
- *If* $\mathcal{C}, N \vdash \mathrm{s} : \mathcal{S} \Rightarrow P$ *then* $N \cap P = \emptyset$*.*

Proof. *A simple rule induction.*

Since the state is intended to record the variables occurring free in the context, we define:

Definition 3.32 (Rigidity). A context $\mathcal{C}$ is *rigid with respect to* N, written $\mathcal{C}, N$ **rigid** if, and only if, $\mathrm{FV}(\mathcal{C}) \subseteq N$.

As long as we start with $\mathcal{C}, N$ **rigid**, as a consequence of Property 3.31, type variables generated during classification will never be confused with types occurring in the context.

To a type theorist, the generative judgements appear odd. The intrusion of the state forces a left-to-right dependency on the order of premises in the typing rules which is a departure from the standard compositional formulation of typing rules in Type Theory. The fact that the type of the term may contain "new" free type variables, that do not occur free in the context, is peculiar (conventional type theories enjoy the free variable property: the type of a term is closed with respect to the variables occurring free in the context). Perhaps for this reason, generativity has developed its own mystique and its own terminology. In the Definition [MTH90, MTH96], type variables are called "names", to stress their persistent, generative nature. Generativity is presented as an extra-logical device, useful for programming language type systems, but distinct from the more traditional type-theoretic constructs, as these can be related to constructive interpretations of logical connectives. In Chapter 4 we will dispel this mystique, reformulating the generative classification judgements in terms of familiar type-theoretic constructs, obtaining more palatable, state-less classification judgements.

3.3.4 The Different Roles of Bound Type Variables

In defining Modules, we have used the notation of Standard ML for semantic structures $(P)\mathcal{S} \in \mathit{Sig}$ and functors $(P)(\mathcal{S}, (Q)\mathcal{S}') \in \mathit{Fun}$. Observe that Standard ML is decidedly non-committal in its choice of binding operators: parentheses are used uniformly to bind sets of type variables. The use of the same notation obscures the different roles that these binding constructs play. In this section, we discuss the purpose of these binding constructs, and propose a more meaningful notation for them. The notation will be adopted in later chapters.

Signatures as Families of Structures

What exactly are signature expressions and the semantic signatures they denote? In the Standard ML literature, signatures are vaguely referred to as the types of structure expressions. However, inspecting the rules we find that the type of a structure is a semantic structure $\mathcal{S}$, while a semantic signature has the form $(P)\mathcal{S}$, binding a set of type variables appearing in the structure $\mathcal{S}$. Let S be a signature expression denoting $(P)\mathcal{S}$, i.e. $\mathcal{C} \vdash \mathrm{S} \triangleright (P)\mathcal{S}$. Each type variable $\alpha^{\mathrm{k}} \in P$ arises from some type specification of the form **type** t : k by an application of Rule (E-2). Since such phrases only specify the kind k of the type component, without determining its definition, it is clear that signatures merely specify families of structures, indexed by the realisations of their bound type variables.

In fact, the role of the signature expression S changes according to the kind of phrase in which it appears. In a functor definition **functor** F (X : S) = s , S specifies that the functor should be uniformly applicable to any argument whose type is in the family $(P)\mathcal{S}$. In a signature curtailment s $\succeq$ S, S is used to check that the type of s is at least as rich as some type that is a member of the family $(P)\mathcal{S}$. The type of the complete phrase is that particular member, which may have less rich components, but by the definition of enrichment, will agree on its type components with the type components of s. In a signature abstraction s $\setminus$ S, S is also used to check that the type of s is at least as rich as some type that is a member of the family $(P)\mathcal{S}$. However, the type of the complete phrase is not that particular member, but a generic member of the family. This is enforced by generating fresh variables for the variables in P.

Given these observations, it should be clear that the denotation of the signature expression is never used as the type of a phrase, but merely as an aid in the construction of a type. The common denominator of all of these usages is the way in which the signature expression serves to specify a family of types. To emphasise this role, we will from now on use a different notation for semantic signatures:

Notation (Semantic Signatures). A semantic signature $(P)\mathcal{S} \in \mathit{Sig}$ is identified with its notational variant $\Lambda P.\mathcal{S}$. Variables in P are bound in $\mathcal{S}$. The signature $\Lambda P.\mathcal{S}$ specifies a family of semantic structures, whose members are obtained by realising type variables in P. The use of Λ as a binder stresses that the bound variables are *parameters* indexing a family of semantic structures.

Since the parameters P of a semantic signature $\Lambda P.\mathcal{S}$ are type variables, and the structure $\mathcal{S}$ is itself a form of type used to classify terms, from a type-theoretic perspective, the signature corresponds to a parameterised type, similar to the notion of parameterised type we encountered in Section 2.2.3.

Functors as Polymorphic Functions

Recall the functor introduction rule (Rule E-17):

$$\frac{\begin{array}{ll}\mathcal{C} \vdash \mathrm{S} \rhd (P)\mathcal{S} & P \cap N = \emptyset \\ \mathcal{C}[\mathrm{X} : \mathcal{S}], N \cup P \vdash \mathrm{s} : \mathcal{S}' \Rightarrow Q & \mathcal{C}[\mathrm{F} : (P)(\mathcal{S}, (Q)\mathcal{S}')], N \vdash \mathrm{b} : \mathcal{S}'' \Rightarrow M\end{array}}{\mathcal{C}, N \vdash \textbf{functor}\ \mathrm{F}\ (\mathrm{X} : \mathrm{S})\ =\ \mathrm{s}\ \textbf{in}\ \mathrm{b} : \mathcal{S}'' \Rightarrow M}$$

From the previous discussion, we can say that the signature expression S denotes the family of structures $\Lambda P.\mathcal{S}$. The functor body s is classified in the context extended with the assumption $[\mathrm{X} : \mathcal{S}]$. Assuming $\mathcal{C}, N$ **rigid**, the second premise ensures that $P \cap \mathrm{FV}(\mathcal{C}) = \emptyset$, hence variables in P are treated as arbitrary, formal type parameters. The semantic functor $(P)(\mathcal{S}, (Q)\mathcal{S}')$, i.e. the type of F, is obtained by discharging first the assumption $[\mathrm{X} : \mathcal{S}]$ from the result of classifying the functor body, and then discharging the type parameters P. In the first step, we obtain a function taking a structure of type $\mathcal{S}$ as an argument. In the second step, we generalise this function on its free type parameters to obtain a *polymorphic* function. The functor elimination rule (Rule (E-21)) reflects the polymorphic behaviour of functors. Before applying the functor, we must first choose a realisation (i.e. an instantiation) of its type parameters. This realisation is used to check that the type of the actual argument is at least as rich as the realisation of the functor's domain. The conventional description of a functor as a function taking structures to structures is clearly inaccurate. A functor is a polymorphic function on structures that first needs to be applied to a type realisation, before it can be applied to an actual argument. Functors are thus similar to the polymorphic functions we encountered in Section 2.2.2. For clarity, the type of a functor should be written using universal quantification over its type parameters.

What of the range $(Q)\mathcal{S}'$ of a semantic functor $(P)(\mathcal{S}, (Q)\mathcal{S}')$? By adopting the notation of semantic signatures, Standard ML suggests that applying a functor to a structure expression returns a term whose type is the *family* of types specified by the signature $(Q)\mathcal{S}'$. This doesn't make sense and overloading the notation in this way is confusing. When a functor is applied, the bound variables of the result signature give rise to fresh generative types. These types are essentially abstract. For a given realisation of its type parameters, applying the functor to a structure expression therefore returns a term whose type is some *member* of the family of types specified by its result signature. Since the bound variables of the result are replaced by generative type variables, precisely which member of this family remains unknown. We emphasise the distinction between functor ranges and semantic signatures by introducing a new notation for functor ranges:

Notation (Existential Structures). An existential structure $\mathcal{X} \in \mathit{ExStr}$ is an existentially quantified structure of the form $\exists P.\mathcal{S}$. Variables in P are bound in $\mathcal{S}$. Intuitively, $\exists P.\mathcal{S}$ is a type used to classify terms. A term belongs to this type if, and only if, there exists a realisation φ of the variables in P such that the term has type $\varphi(\mathcal{S})$. In other words, a term belongs to this type if, and only if, its type is some member of the family of types $\Lambda P.\mathcal{S}$.

We can stress the observations of the preceding discussion by changing the notation of semantic functors.

Notation (Semantic Functors). A semantic functor $(P)(\mathcal{S},(Q)\mathcal{S}') \in \mathit{Fun}$ is identified with its notational variant $\forall P.\mathcal{S} \to \exists Q.\mathcal{S}'$. $\forall P.\mathcal{S} \to \exists Q.\mathcal{S}'$ is a type classifying a functor. The use of $\forall$ as the outermost binder stresses that the bound variables are universally quantified: the functor is polymorphic. The $\to$ stresses that realising the functor's type parameters yields a function on terms. The existential quantifier stresses that, for any realisation φ of the functor's type parameters (i.e. $\mathrm{Dom}(\varphi) = P$), applying the functor returns a term with existential structure $\varphi\,(\exists Q.\mathcal{S}')$, in other words, a term whose type is some member of the family of structures $\varphi\,(\Lambda Q.\mathcal{S}')$.

3.3.5 Is Mini-SML dependently typed?

We can now return to answer the question of whether Mini-SML's syntactic dependency of type phrases on term phrases actually induces a semantic dependency of types on terms. Given the considerable effort that has been invested in finding dependently typed models of Standard ML, it is surprising to find that the underlying semantic objects reveal no dependency on terms. Perhaps the best way to see this is to consider the semantic objects assigned to phrases exhibiting syntactic dependencies.

Consider the signature expression in Figure 3.17(a). Notice the syntactic

```
sig structure X : sig type t:0
                  end;
    structure Y : sig type u:0;
                      type v = X.t → u
                  end;
    val y: X.t → Y.v
end
```

(a) The signature suggests a dependency of types on terms.

$$\begin{aligned}\Lambda\{\alpha,\beta\}.\ (\mathbf{X} &: (\mathbf{t} = \alpha),\\ \mathbf{Y} &: (\mathbf{u} = \beta,\\ &\qquad \mathbf{v} = \alpha \to \beta),\\ \mathbf{y} &: \alpha \to \alpha \to \beta)\end{aligned}$$

(b) The signature's non-dependent denotation.

Figure 3.17: A term-dependent signature expression and its non-dependent denotation.

dependency of the specifications of **v** and **y** on the structure identifiers **X** and **Y**. In fact, we can be a little more specific, by observing that the definition of **v** and the specification of **y** depend on type components **X**.**t**, **u**, and **Y**.**v**. The phrase denotes the semantic signature in Figure 3.17(b). Notice how the denotations of **t** and **u** are represented by type variables α and β. In the semantic signature, syntactic dependencies on **X**.**t**, **u**, and **Y**.**v** have been systematically replaced by semantic occurrences of the second-order variables α and β. In particular, the syntactic dependencies on the term phrases **X** and **Y** have been removed. The simplification is effected by the denotation judgements, which, by assigning type variables to type specifications, manage to replace all type occurrence phrases by their denotations. Observe that the identifiers appearing within semantic structures are not bound in any way, they are merely tags like the field names of record types.

As another example, consider the functor in Figure 3.18(a). It returns a type **w** whose definition is syntactically dependent on the functor argument **Z**. Figure 3.18(b) shows the semantic functor classifying **F**. Again the syntactic dependence of the result type **w** on **Z**.**X**.**t** and **Z**.**Y**.**v** has been simplified by replacing these type occurrence phrases by their denotations. The syntactic term dependency on **Z** is completely removed, leaving a residual second-order dependency on the functor's type parameters α and β.

In summary, the *first-order* dependencies of types on terms, apparent in the syntax of Mini-SML, boil down to *second-order* dependencies of semantic objects on type variables.

3.4 Conclusion

In Section 3.1, we defined the Modules language. The definition of this language is based directly on the definition of Standard ML Modules, capturing its essential features. Our definition of Modules is parameterised by an arbitrary Core language. For concreteness, in Section 3.2 we presented a particular instance of the Core language: Core-ML. Core-ML supports both parameterised type and polymorphic values, capturing the two main features of Standard ML's Core language that are relevant to the definition of Modules. In Section 3.3, we defined Mini-SML as the language obtained by combining the definitions of Modules and Core-ML and proceeded with an informal analysis of the type-theoretic underpinnings of Mini-SML. We discussed how the syntactic dependency of types on terms is suggestive of a dependently-typed language. Indeed, this feature has prompted many

```
functor F(Z : sig structure X : sig type t:0
                                end;
                  structure Y : sig type u:0;
                                    type v = X.t → u
                                end;
                  val y: X.t → Y.v
                end ) =
        struct type w = Z.X.t → Z.Y.v;
               val z = Z.y
        end
in ...
```

(a) The functor body suggests a dependency of types on terms.

$$
\begin{aligned}
\forall\{\alpha,\beta\}.(&\mathbf{X} : (\mathbf{t} = \alpha),\\
&\mathbf{Y} : (\mathbf{u} = \beta,\\
&\qquad \mathbf{v} = \alpha \to \beta),\\
&\mathbf{y} : \alpha \to \alpha \to \beta)\\
&\to \exists\emptyset.(\mathbf{w} = \alpha \to \alpha \to \beta,\\
&\qquad \mathbf{z} : \alpha \to \alpha \to \beta)
\end{aligned}
$$

(b) The functor's non-dependent type.

Figure 3.18: A dependent functor and its non-dependent type.

researchers in the area to propose that dependent types underly the type structure of Standard ML. However, by inspecting the static semantics of Mini-SML, we have found ample evidence to suggest that Mini-SML can be understood by relying only on the simpler type-theoretic notions of type parameterisation, type quantification and subtyping. Since Mini-SML is merely a cut down version of Standard ML, we can make the counter-claim that dependent types have no role in the static semantics of Standard ML.

The generative classification judgements, however, do not sit nicely with our type-theoretic understanding of other aspects of the static semantics. In Chapter 4, we focus our attention on the generative judgements, and expose them as a particularly operational incarnation of existential quantification over types. By the end of that chapter, we hope to have discredited the claim that dependent types are necessary to explain the type structure of Standard ML.

Chapter 4

Type Generativity as Existential Quantification

In this chapter, we present a new static semantics for structure bodies and expressions. It is intended as a more type-theoretic alternative to the generative classification judgements we gave in Chapter 3. Our main objective is to first explain and then eliminate the state of type variables maintained by the generative classification rules. Considerable effort is devoted to proving the equivalence of the two systems. We claim that the resulting system is more type-theoretic in style and easier to understand. In later chapters, we will substantiate this claim by using the alternative semantics as the basis for significant extensions to the language.

The chapter is organised as follows. In Section 4.1, we give an informal account of the role generativity plays in the existing semantics. In Section 4.2, we suggest an alternative semantics for generative phrases. We first introduce the concept of existential structures. We then define judgements classifying structure bodies and expressions by existential structures. Unlike their generative counterparts, these judgements are state-less and have the more familiar form of typing judgements. Section 4.3 is devoted to proving that the generative classification judgements and their state-less replacements are equivalent. After setting up the necessary machinery, we first prove an easy completeness result: every generative classification (of interest) gives rise to a state-less classification. We then prove a technically more difficult soundness result: every state-less classification (of interest) gives rise to a generative classification. The proof of soundness factors into two parts. In the first, we prove a generalised induction principle for state-less classification judgements. In the second, we use this induction principle to prove our result. Section 4.4 briefly discusses the implications of our equivalence result.

The work in this chapter is a slight generalisation of the author's earlier work in [Rus96], which presents similar results for a simplified, core-less Modules language.

Remark 4.0.1. We adopt the notational changes to semantic signatures and functors motivated in Section 3.3.4. That is, from now on we will abandon the Standard ML notation and write a semantic signature $\mathcal{L} \in \mathit{Sig}$ in the form $\mathcal{L} \equiv \Lambda P.\mathcal{S}$, to stress that it is a *parameterised* structure, and write a semantic functor $\mathcal{F} \in \mathit{Fun}$ in the form $\mathcal{F} \equiv \forall P.\mathcal{S} \to \mathcal{X}$ or $\mathcal{F} \equiv \forall P.\mathcal{S} \to \exists Q.\mathcal{S}'$ to stress that it is the type of a *polymorphic* function on structures with an *existentially quantified* range.

4.1 Generativity: An Informal Account

In Section 3.1.3 we presented the generative classification judgements used to determine the types of structure bodies and expressions. In Section 3.3.3 we briefly discussed the way in which the rules of these judgements maintain a state of generated types during classification. The sense in which a generative judgement returns new types was captured by Property 3.31 (Generativity). Definition 3.32 (Rigidity) captured a pre-condition on the initial context and state that is necessary[1] to ensure that the generative rules are sound. This condition is maintained as an invariant during classification.

The following sections provide an informal account of why "generativity" is needed at all. In a nutshell, generativity is used to avoid the unsafe identification of types that might otherwise lead to run-time type errors.

4.1.1 The Rationale for Distinguishing Syntactic Type Identifiers From Semantic Type Variables

In a naive semantics, one might choose to identify syntactic type identifiers with the semantic type variables they denote. Unfortunately, since Mini-SML, like Standard ML, allows the redefinition of syntactic type identifiers, this can lead to the confusion of types that ought to be kept distinct.

Consider the phrase in Figure 4.1(a). In the naive semantics it would "type-check". However, this is clearly not safe, since it leads, at run-time, to the application of 1 to 2. The phrase is not sound. Initially, **x** is used to coerce 1 to a value of type **t**. Here, the first defining occurrence of **t** is intended.

[1]but, as we shall see, not quite sufficient.

$$\begin{array}{l}
\textbf{datatype t} \ = \ \textbf{int with x}, \textbf{y}; \\
\textbf{structure X} = \textbf{struct datatype t} \ = \ \textbf{int} \rightarrow \textbf{int with x}', \textbf{y}'; \\
\qquad\qquad\qquad\qquad \textbf{val z} = (\textbf{y}' \ (\textbf{x} \ 1)) \ 2 \\
\qquad\qquad\quad \textbf{end}
\end{array}$$

(a) The phrase is not sound: it attempts to masquerade 1 as a function and apply it to 2, resulting in a run-time error. The phrase should be rejected by a sound static semantics.

$$\begin{array}{l}
\textbf{datatype t}_{\textbf{t}} \ = \ \textbf{int with x}, \textbf{y}; \\
\textbf{structure X} = \textbf{struct datatype t}_{\textbf{t}} \ = \ \textbf{int} \rightarrow \textbf{int with x}', \textbf{y}'; \\
\qquad\qquad\qquad\qquad \textbf{val z} = (\textbf{y}'_{\textbf{t}\rightarrow\textbf{int}\rightarrow\textbf{int}}(\textbf{x}_{\textbf{int}\rightarrow\textbf{t}} \ 1)_{\textbf{t}} \ )_{\textbf{int}\rightarrow\textbf{int}} \ 2 \\
\qquad\qquad\quad \textbf{end}
\end{array}$$

(b) An unsound classification of the same phrase constructed in a naive semantics that identifies syntactic type identifiers with semantic type variables.

$$\begin{array}{l}
\textbf{datatype t}_{\alpha} \ = \ \textbf{int with x}, \textbf{y}; \\
\textbf{structure X} = \textbf{struct datatype t}_{\beta} \ = \ \textbf{int} \rightarrow \textbf{int with x}', \textbf{y}'; \\
\qquad\qquad\qquad\qquad \textbf{val z} = \underline{(\textbf{y}'_{\beta\rightarrow\textbf{int}\rightarrow\textbf{int}}(\textbf{x}_{\textbf{int}\rightarrow\alpha} \ 1)_{\alpha} \ )} \ 2 \\
\qquad\qquad\quad \textbf{end}
\end{array}$$

(c) The unsuccessful but sound classification of the same phrase according to the semantics in Chapter 3. The offending subphrase is underlined.

Figure 4.1: An unsound phrase illustrating the purpose of distinguishing between syntactic type identifiers and semantic type variables.

Now $\mathbf{y}'$ allows us to coerce any value of type $\mathbf{t}$, where the *second* defining occurrence of $\mathbf{t}$ is intended, to a value of function type $\mathbf{int} \rightarrow \mathbf{int}$. Unfortunately, by sharing their denotations, the naive semantics identifies both definitions and fails to catch the type violation in the expression $\mathbf{y}'\ (\mathbf{x}\ 1)$. In our semantics, we avoid this pitfall by assigning distinct type variables α and β to the first and second defining occurrences of $\mathbf{t}$. The situation is summarised in Figures 4.1(b) and 4.1(c). In each, the defining occurrences are annotated with their semantic representations and key subphrases are annotated with their types.

Of course, if we rule out the redefinition of type identifiers already bound in the context, the need to distinguish them using type variables disappears. But this is a harsh restriction and goes against one of the main motivations for Modules: to provide a mechanism of name space control allowing component names to be reused within different modules. After all, outside of the structure body, the definitions are referred to by distinct phrases $\mathbf{t}$ and $\mathbf{X}.\mathbf{t}$.

4.1.2 The Rationale for Maintaining a State of Generated Type Variables

Although the example of Section 4.1.1 illustrates how type variables are used to distinguish between different definitions of the same type identifier, it does not explain the need to maintain a *state* of type variables throughout classification. In a typical classification, we must, at certain points, choose suitably “fresh” variables to represent syntactic type identifiers. In our example, we had to choose β to be fresh for α. One may be tempted to define “fresh” to simply mean “distinct from the variables free in the current context”, as this definition correctly deals with our example yet does away with the overhead of the state. Unfortunately, in the general case, this condition is too weak to ensure soundness. Indeed, classification must maintain a state of all variables generated *so far*, not just those visible in the current context. The following example shows why.

Consider the phrase in Figure 4.2(a). As in the example of Section 4.1.1, a sound semantics should reject the phrase since it leads to a run-time error in the definition of $\mathbf{z}$.

Suppose that, using a putatively simpler, state-less semantics, we were only to require that the type variables chosen for $\mathbf{t}$ and $\mathbf{u}$ be distinct from the variables currently free in the context of their respective definitions. Figure 4.2(b) shows what can go wrong. We indicate, at the beginning of each structure body b, the set N of variables free in the local context, using

$$\begin{array}{l}
\textbf{structure X} = \textbf{struct datatype t} \;=\; \textbf{int with x}, \textbf{y end};\\
\textbf{structure Y} = \textbf{struct structure X} \;=\; \textbf{struct}\;\; \textbf{end};\\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{datatype u} \;=\; \textbf{int} \rightarrow \textbf{int with x}', \textbf{y}'\\
\qquad\qquad\quad \textbf{end};\\
\textbf{val z} = (\textbf{Y}.\textbf{y}'\ (\textbf{X}.\textbf{x}\ 1))\ 2
\end{array}$$

(a) The phrase is unsound, attempting to apply 1 to 2. It should be rejected by a sound static semantics.

$$\begin{array}{l}
\overset{\emptyset}{\lceil}\textbf{structure X} = \textbf{struct}\ \overset{\emptyset}{\lceil}\textbf{datatype t}_{\alpha} \;=\; \textbf{int with x}, \textbf{y end};\\
\overset{\{\alpha\}}{\lceil}\textbf{structure Y} = \textbf{struct}\ \overset{\{\alpha\}}{\lceil}\textbf{structure X} \;=\; \textbf{struct}\;\; \textbf{end};\\
\qquad\qquad\qquad\qquad\qquad\quad \overset{\emptyset}{\lceil}\textbf{datatype u}_{\alpha} \;=\; \textbf{int} \rightarrow \textbf{int with x}', \textbf{y}'\\
\qquad\qquad\quad \textbf{end};\\
\overset{\{\alpha\}}{\lceil}\textbf{val z} = (\textbf{Y}.\textbf{y}'_{\alpha\rightarrow\textbf{int}\rightarrow\textbf{int}}(\textbf{X}.\textbf{x}_{\textbf{int}\rightarrow\alpha}\ 1)_{\alpha}\ )_{\textbf{int}\rightarrow\textbf{int}}\ 2
\end{array}$$

(b) An unsound classification of the phrase in Figure 4.2(a) resulting from a state-less semantics.

$$\begin{array}{l}
\overset{\emptyset}{\downarrow}\textbf{structure X} = \textbf{struct}\ \overset{\emptyset}{\downarrow}\textbf{datatype t}_{\alpha} \;=\; \textbf{int with x}, \textbf{y}\overset{\{\alpha\}}{\uparrow}\ \textbf{end};\\
\overset{\{\alpha\}}{\downarrow}\textbf{structure Y} = \textbf{struct}\ \overset{\{\alpha\}}{\downarrow}\textbf{structure X} \;=\; \textbf{struct}\;\; \textbf{end};\\
\qquad\qquad\qquad\qquad\qquad\quad \overset{\{\alpha\}}{\downarrow}\textbf{datatype u}_{\beta} \;=\; \textbf{int} \rightarrow \textbf{int with x}', \textbf{y}'\overset{\{\beta\}}{\uparrow}\\
\qquad\qquad\quad \textbf{end};\\
\overset{\{\alpha,\beta\}}{\downarrow}\textbf{val z} = \underline{(\textbf{Y}.\textbf{y}'_{\beta\rightarrow\textbf{int}\rightarrow\textbf{int}}(\textbf{X}.\textbf{x}_{\textbf{int}\rightarrow\alpha}\ 1)_{\alpha}\ )}\ 2
\end{array}$$

(c) A partial, correctly unsuccessful classification of the phrase in Figure 4.2(a). The state prevents the offending subphrase from being accepted. The type violation is underlined.

Figure 4.2: An unsound phrase illustrating the necessity of maintaining a state of type variables.

the notation $\overset{N}{\lceil}$ b. In addition, defining occurrences of **t** and **u** are annotated with their semantic representations and key subphrases with their types. The problem is that **t** and **u** are assigned the same type variable α, even though they must be distinguished. The problem arises because α, already set aside for **t**, no longer occurs free in the context by the time we need to choose a fresh variable for **u**: it is eclipsed by the shadow of the second definition of **X**. Thus we may again pick α and incorrectly accept the definition of **z**.

In Figure 4.2(c) we can see how the use of a state maintains soundness. We indicate, at the beginning of each structure body b, the state N of variables generated so far, and, at its end, the variables M generated during its classification. We use the notation $\overset{N}{\downarrow}$ b $\overset{M}{\uparrow}$, corresponding to a classification $\ldots, N \vdash \mathrm{b} : \ldots \Rightarrow M$. Observe that generated variables are accumulated in the state as we traverse the phrase. At the definition of **u**, α is recorded in the state, even though it no longer occurs free in the context, forcing the choice of a distinct variable β. In turn, this leads to the detection of the type violation.

4.1.3 The Rationale for Generative Functor Application

Finally, let us examine the relationship between functors and type generativity. Consider a typical functor definition **functor** F (X : S) = s **in** b. Recall that classifying the body s of F with respect to its formal argument X may itself generate new types. By way of comparison, in the structure definition **structure** X = s;b, these variables would be added to the state before proceeding with the classification of b. In the case of a functor definition, however, they are *not* added to the state. Instead, they become (existentially) bound in the result structure of F. At each application of F to an actual argument, fresh variables are generated to replace them in the actual result. In effect, generativity is delayed from the point of definition to the point of application, each application producing a fresh set of types.

Consider the misleading example in Figure 4.3(a). The phrase does not type-check. In Figure 4.3(b) we have annotated the phrase with semantic objects to show why classification fails: the types **Y**.**u** and **Z**.**u** are incompatible. Each application of **F** has generated a fresh type, δ and γ respectively, to replace β.

At first glance, this seems overly restrictive since the example is, in fact, sound: the evaluation of **z** does not cause a run-time error. Could we not, as in Figure 4.3(c), safely generate types once and for all at the functor's

$$\begin{array}{l}
\mathbf{functor}\ \mathbf{F}\ (\mathbf{X} : \mathbf{sig}\ \ \mathbf{end})\ =\ \mathbf{struct}\ \mathbf{datatype}\ \mathbf{u}\ =\ \mathbf{int}\ \mathbf{with}\ \mathbf{x}, \mathbf{y}\ \mathbf{end}\\
\mathbf{in}\\
\mathbf{structure}\ \mathbf{Y}\ =\ \mathbf{F}\ (\mathbf{struct}\ \ \mathbf{end});\\
\mathbf{structure}\ \mathbf{Z}\ =\ \mathbf{F}\ (\mathbf{struct}\ \ \mathbf{end});\\
\mathbf{val}\ \mathbf{z} = \mathbf{Z}.\mathbf{y}\ (\mathbf{Y}.\mathbf{x}\ 1)
\end{array}$$

(a) The phrase is sound but fails to type-check.

$$\begin{array}{l}
\mathbf{functor}\ \lfloor \mathbf{F}\ (\mathbf{X} : \mathbf{sig}\ \ \mathbf{end})\ =\ \mathbf{struct}\ \mathbf{datatype}\ \mathbf{u}\ =\ \mathbf{int}\ \mathbf{with}\ \mathbf{x}, \mathbf{y}\ \mathbf{end}_{\forall\emptyset.\epsilon_{\mathcal{S}}\to\exists\{\beta\}.(\mathbf{u}=\beta,\mathbf{x}:\mathbf{int}\to\beta,\mathbf{y}:\beta\to\mathbf{int})}\\
\mathbf{in}\\
\mathbf{structure}\ \lfloor \mathbf{Y}\ =\ \mathbf{F}\ (\mathbf{struct}\ \ \mathbf{end})_{(\mathbf{u}=\delta,\mathbf{x}:\mathbf{int}\to\delta,\mathbf{y}:\delta\to\mathbf{int})};\\
\mathbf{structure}\ \lfloor \mathbf{Z}\ =\ \mathbf{F}\ (\mathbf{struct}\ \ \mathbf{end})_{(\mathbf{u}=\gamma,\mathbf{x}:\mathbf{int}\to\gamma,\mathbf{y}:\gamma\to\mathbf{int})};\\
\mathbf{val}\ \mathbf{z} = \underline{\mathbf{Z}.\mathbf{y}_{\gamma\to\mathbf{int}}\ (\mathbf{Y}.\mathbf{x}_{\mathbf{int}\to\delta}\ 1)_{\delta}}
\end{array}$$

(b) The unsuccessful classification of the same phrase. The example illustrates the generation of fresh types at each and every functor application. The offending sub-phrase is underlined.

$$\begin{array}{l}
\mathbf{functor}\ \lfloor \mathbf{F}\ (\mathbf{X} : \mathbf{sig}\ \ \mathbf{end})\ =\ \mathbf{struct}\ \mathbf{datatype}\ \mathbf{u}\ =\ \mathbf{int}\ \mathbf{with}\ \mathbf{x}, \mathbf{y}\ \mathbf{end}_{\forall\emptyset.\epsilon_{\mathcal{S}}\to(\mathbf{u}=\beta,\mathbf{x}:\mathbf{int}\to\beta,\mathbf{y}:\beta\to\mathbf{int})}\\
\mathbf{in}\\
\mathbf{structure}\ \lfloor \mathbf{Y}\ =\ \mathbf{F}\ (\mathbf{struct}\ \ \mathbf{end})_{(\mathbf{u}=\beta,\mathbf{x}:\mathbf{int}\to\beta,\mathbf{y}:\beta\to\mathbf{int})};\\
\mathbf{structure}\ \lfloor \mathbf{Z}\ =\ \mathbf{F}\ (\mathbf{struct}\ \ \mathbf{end})_{(\mathbf{u}=\beta,\mathbf{x}:\mathbf{int}\to\beta,\mathbf{y}:\beta\to\mathbf{int})};\\
\mathbf{val}\ \mathbf{z} = \mathbf{Z}.\mathbf{y}_{\beta\to\mathbf{int}}\ (\mathbf{Y}.\mathbf{x}_{\mathbf{int}\to\beta}\ 1)_{\beta}
\end{array}$$

(c) A successful and sound classification of the same phrase in a simpler semantics with non-generative functor application.

Figure 4.3: A sound phrase that nevertheless fails to type-check due to the generativity of functor application. The example misleadingly suggests that generative functor application is overly conservative.

definition? We would add β as a new variable to the state, avoid the overhead of generativity at each application, and accept the innocent phrase as it stands. A simpler notion of semantic functor, minus the set of generative type variables, would do. These modifications yield *applicative* functors, in the sense that every application of a given functor yields equivalent abstract types, instead of generating fresh ones. The idea of an applicative semantics for functors originates with Leroy [Ler95]. Unfortunately, our approach to an applicative semantics is too naive and fails to be sound. The following counter-example illustrates the problem.

Consider the phrase in Figure 4.4. It differs only slightly from the previous one, by introducing a dependency of **u** on the formal argument **X**: in particular, the definition of **u** is a function of the type parameter **X**.**t**. Now, with each application of **F**, the realisation of **u** may vary according to **F**'s actual argument. Indeed, in our example it does: the definition of **z** is unsound and should be rejected. Generating fresh types at each application of **F** is a sufficient guard against violations such as this one (see Figure 4.4(b)). Generating variables once-and-for-all at a functor's definition, as in our naive applicative semantics, is not (see Figure 4.4(c)).

Remark 4.1.1 *(Applicative Functors and Soundness).* Leroy's semantics of applicative functors [Ler95] is different from the naive semantics considered here and does maintain soundness. His semantics is sound, because the abstract types returned by a functor application are expressed as a function of the functor's actual argument. This leads to an applicative behaviour in the sense that two applications of the same functor to the same argument result in equivalent abstract types, while an application of the same functor to a different argument yields distinct abstract types. For instance, if **F** is the functor in Figure 4.4, and **A** and **B** are structure identifiers defined as the two arguments of **F** in Figure 4.4, then each application **F A** returns the same abstract type (**F A**).**u**, but the application **F B** returns a different abstract (**F B**).**u**: the abstract types (**F A**).**u** and (**F B**).**u** are distinct, because their constituent paths (**F B**) and (**F A**) are different (recall that Leroy extends paths to allow applications of functors paths to argument paths). Note, however, that Leroy's syntactic restriction to paths means that this technique can only be applied to applications of functor *paths* to argument *paths*. For instance, this restriction means that the abstract types returned by two applications of **F** to the same *anonymous* structure (**struct type t** = **int end**) will be distinct, even though it is perfectly sound for them to be equivalent.

In Chapter 5, we will give an applicative semantics for functors that

```
functor F(X: sig type t : 0 end) =
        struct datatype u = X.t with x, y end
in
structure Y = F (struct type t = int end);
structure Z = F (struct type t = int → int end);
val z = (Z.y (Y.x 1)) 2
```

(a) The phrase is unsound, attempting to apply 1 to 2. It should be rejected by a sound static semantics.

$$\begin{array}{l}
\textbf{functor } \lfloor \mathbf{F}_{\forall\{\alpha\}.(\mathbf{t}=\alpha)\to\exists\{\beta\}.(\mathbf{u}=\beta,\mathbf{x}:\alpha\to\beta,\mathbf{y}:\beta\to\alpha)}\ (\mathbf{X}\text{: }\textbf{sig type t} : 0\ \textbf{end}) = \\
\qquad \textbf{struct datatype u} = \mathbf{X.t}\ \textbf{with}\ \mathbf{x}, \mathbf{y}\ \textbf{end} \\
\textbf{in} \\
\textbf{structure } \lfloor \mathbf{Y}_{(\mathbf{u}=\delta,\mathbf{x}:\mathbf{int}\to\delta,\mathbf{y}:\delta\to\mathbf{int})} = \mathbf{F}\ (\textbf{struct type t} = \textbf{int end}); \\
\textbf{structure } \lfloor \mathbf{Z}_{(\mathbf{u}=\gamma,\mathbf{x}:(\mathbf{int}\to\mathbf{int})\to\gamma,\mathbf{y}:\gamma\to(\mathbf{int}\to\mathbf{int}))} = \mathbf{F}\ (\textbf{struct type t} = \textbf{int} \to \textbf{int end}); \\
\textbf{val z} = \underline{(\mathbf{Z.y}_{\gamma\to(\mathbf{int}\to\mathbf{int})}\ (\mathbf{Y.x}_{\mathbf{int}\to\delta}\ 1)_{\delta})}\ 2
\end{array}$$

(b) The unsuccessful but sound classification of the same phrase. The example demonstrates how the generation of fresh types at each and every functor application preserves soundness. The denotations of **Y.u** and **Z.u** are correctly distinguished. The offending subphrase is underlined.

$$\begin{array}{l}
\textbf{functor } \lfloor \mathbf{F}_{\forall\{\alpha\}.(\mathbf{t}=\alpha)\to(\mathbf{u}=\beta,\mathbf{x}:\alpha\to\beta,\mathbf{y}:\beta\to\alpha)}\ (\mathbf{X}\text{: }\textbf{sig type t} : 0\ \textbf{end}) = \\
\qquad \textbf{struct datatype u} = \mathbf{X.t}\ \textbf{with}\ \mathbf{x}, \mathbf{y}\ \textbf{end} \\
\textbf{in} \\
\textbf{structure } \lfloor \mathbf{Y}_{(\mathbf{u}=\beta,\mathbf{x}:\mathbf{int}\to\beta,\mathbf{y}:\beta\to\mathbf{int})} = \mathbf{F}\ (\textbf{struct type t} = \textbf{int end}); \\
\textbf{structure } \lfloor \mathbf{Z}_{(\mathbf{u}=\beta,\mathbf{x}:(\mathbf{int}\to\mathbf{int})\to\beta,\mathbf{y}:\beta\to(\mathbf{int}\to\mathbf{int}))} = \mathbf{F}\ (\textbf{struct type t} = \textbf{int} \to \textbf{int end}); \\
\textbf{val z} = (\mathbf{Z.y}_{\beta\to(\mathbf{int}\to\mathbf{int})}\ (\mathbf{Y.x}_{\mathbf{int}\to\beta}\ 1)_{\beta})_{\mathbf{int}\to\mathbf{int}}\ 2
\end{array}$$

(c) A successful but unsound classification of the same phrase constructed in a naive semantics with applicative functors. By sharing the same denotation, **Y.u** and **Z.u** are incorrectly identified.

Figure 4.4: An unsound phrase illustrating the need for generative functor application.

is sound and also works for anonymous arguments, but is slightly different from Leroy's notion. In our proposal, two applications of the same functor to equivalent type arguments (i.e. an equivalent realisation) yield equivalent abstract types, while an application of the same functor to distinct type arguments yields distinct abstract types. We will compare our semantics with Leroy's in Chapter 9.

4.2 A Type-Theoretic Semantics

In this section we present an alternative static semantics for structure bodies and expressions. The idea is to replace the mysterious reliance on a state of generated types with a well understood construct from type theory: existential quantification over types. The intuition arises from a slight shift in perspective. Instead of treating the classification of structure bodies and expressions as resulting in a semantic structure $\mathcal{S}$ with the possible side-effect of generating new types P, we shall treat the new types as an integral part of the classifying object.

We first define:

Definition 4.1 (Existential Structures). An existential structure $\mathcal{X} \in$ *ExStr* is an existentially quantified structure of the form $\exists P.\mathcal{S}$. Variables in P are bound in $\mathcal{S}$. Intuitively, $\exists P.\mathcal{S}$ is a type used to classify terms. A term belongs to this type if, and only if, there exists a realisation φ of the variables in P such that the term has type $\varphi(\mathcal{S})$. In other words, a term belongs to this type if, and only if, its type is some member of the family of types $\Lambda P.\mathcal{S}$.

In the new semantics, structure bodies and expressions are classified by *existential structures* $\mathcal{X} \in$ *ExStr*. Adopting this view allows us to replace the state-full judgements:

$$\mathcal{C}, N \vdash \mathrm{b} : \mathcal{S} \Rightarrow P$$

$$\mathcal{C}, N \vdash \mathrm{s} : \mathcal{S} \Rightarrow P$$

by the state-less judgements:

$$\mathcal{C} \vdash \mathrm{b} : \exists P.\mathcal{S}$$

$$\mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S}.$$

The key idea is to replace *global* generativity with respect to a state by the implicit introduction and *local* elimination of existential quantifiers.

$$\mathcal{C} \vdash \mathrm{b} : \mathcal{X}$$
In context $\mathcal{C}$, structure body b has existential structure $\mathcal{X}$.
$$\mathcal{C} \vdash \mathrm{s} : \mathcal{X}$$
In context $\mathcal{C}$, structure expression s has existential structure $\mathcal{X}$.

Figure 4.5: State-less Classification Judgements for Structure Bodies and Expressions

4.2.1 Static Semantics

The new static semantics of structure bodies and expressions is defined by the judgements in Figure 4.5. We have indicated, below each judgement, its intended English reading. The judgements are defined by the following rules. Instead of relying on a global state, the rules employ side conditions on sets of bound variables. The side conditions prevent the capture of free variables in the usual way. The identification of existential structures up to capture-avoiding renamings of bound variables means that we can always rename bound variables as necessary to satisfy the side conditions.

Structure Bodies $\boxed{\mathcal{C} \vdash \mathrm{b} : \mathcal{X}}$

$$\frac{\begin{array}{cc} \mathcal{C} \vdash \mathrm{d} \triangleright d & P \cap \mathrm{FV}(d) = \emptyset \\ \mathcal{C}[\mathrm{t} = d] \vdash \mathrm{b} : \exists P.\mathcal{S} & \mathrm{t} \notin \mathrm{Dom}(\mathcal{S}) \end{array}}{\mathcal{C} \vdash \mathbf{type}\ \mathrm{t} = \mathrm{d}; \mathrm{b} : \exists P.\mathrm{t} = d, \mathcal{S}} \tag{T-13}$$

(T-13) The type definition can be classified provided d denotes definable type d and b has existential structure $\exists P.\mathcal{S}$ in the extended context. Ensuring free variables in d are not accidentally captured by bound variables in P, we implicitly eliminate the existential $\exists P.\mathcal{S}$, extend $\mathcal{S}$ to record the denotation of the type component t and then hide the hypothetical types by existentially quantifying over the resulting structure.

$$\frac{\begin{array}{cc} \mathcal{C} \vdash \mathrm{e} : v & P \cap \mathrm{FV}(v) = \emptyset \\ \mathcal{C}[\mathrm{x} : v] \vdash \mathrm{b} : \exists P.\mathcal{S} & \mathrm{x} \notin \mathrm{Dom}(\mathcal{S}) \end{array}}{\mathcal{C} \vdash \mathbf{val}\ \mathrm{x} = \mathrm{e}; \mathrm{b} : \exists P.\mathrm{x} : v, \mathcal{S}} \tag{T-14}$$

(T-14) The value definition can be classified provided e has type v and b has existential structure $\exists P.\mathcal{S}$ in the extended context. Ensuring free variables in v are not accidentally captured by bound variables in P, we implicitly eliminate the existential $\exists P.\mathcal{S}$, extend $\mathcal{S}$ to record the type of the value component x and then hide the hypothetical types by existentially quantifying over the resulting structure.

$$\frac{\begin{array}{ll} \mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S} & P \cap \mathrm{FV}(\mathcal{C}) = \emptyset \\ \mathcal{C}[\mathrm{X} : \mathcal{S}] \vdash \mathrm{b} : \exists P'.\mathcal{S}' & \\ P' \cap (P \cup \mathrm{FV}(\mathcal{S})) = \emptyset & \mathrm{X} \notin \mathrm{Dom}(\mathcal{S}') \end{array}}{\mathcal{C} \vdash \textbf{structure}\ \mathrm{X}\ =\ \mathrm{s;b} : \exists P \cup P'.\mathrm{X} : \mathcal{S}, \mathcal{S}'} \qquad \text{(T-15)}$$

(T-15) Assume s has existential structure $\exists P.\mathcal{S}$. To provide access to the components of s, we locally eliminate the existential, introducing fresh hypothetical types P, and type check b in the suitably extended context to obtain a semantic structure $\exists P'.\mathcal{S}'$. Now $\exists P'.\mathcal{S}'$ may contain occurrences of the locally eliminated types in P, and these should not escape their scope: so we eliminate the existential $\exists P'.\mathcal{S}'$, extend the structure $\mathcal{S}'$ by the component X and existentially quantify over the types $P \cup P'$, yielding the result type $\exists P \cup P'.\mathbf{X} : \mathcal{S}, \mathcal{S}'$. The quantification over P in this type means that the variables in P do not escape their scope. The first side condition of the rule ensures that the variables in P are treated as hypothetical types and not confused with any existing types in $\mathcal{C}$. The second side condition prevents the accidental capture of variables in P and $\mathcal{S}$ by bound variables in P'.

$$\frac{\begin{array}{ll} \mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S} & \mathcal{C}[\mathrm{X} : \mathcal{S}] \vdash \mathrm{b} : \exists P'.\mathcal{S}' \\ P \cap \mathrm{FV}(\mathcal{C}) = \emptyset & P' \cap P = \emptyset \end{array}}{\mathcal{C} \vdash \textbf{local}\ \mathrm{X}\ =\ \mathrm{s}\ \textbf{in}\ \mathrm{b} : \exists P \cup P'.\mathcal{S}'} \qquad \text{(T-16)}$$

(T-16) Similar to Rule (T-15) except that X does not become a component of the resulting structure.

$$\frac{\begin{array}{l} \mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{S} \\ P \cap \mathrm{FV}(\mathcal{C}) = \emptyset \\ \mathcal{C}[\mathrm{X} : \mathcal{S}] \vdash \mathrm{s} : \mathcal{X}' \\ \mathcal{C}[\mathrm{F} : \forall P.\mathcal{S} \to \mathcal{X}'] \vdash \mathrm{b} : \mathcal{X} \end{array}}{\mathcal{C} \vdash \textbf{functor}\ \mathrm{F}\ (\mathrm{X} : \mathrm{S})\ =\ \mathrm{s}\ \textbf{in}\ \mathrm{b} : \mathcal{X}} \qquad \text{(T-17)}$$

(T-17) The signature expression S denotes a family of semantic structures, $\Lambda P.\mathcal{S}$. We want F to be applicable to any argument whose type matches the signature $\Lambda P.\mathcal{S}$. To this end, we classify the body s of F in the context extended with the assumption that the formal argument X has type $\mathcal{S}$. Because we ensure that P is a locally fresh set of variables, the type $\mathcal{S}$ is a generic instance of $\Lambda P.\mathcal{S}$. The classification of s is an existentially quantified structure $\mathcal{X}'$, which may contain occurrences of our generic variables P. Intuitively, since the functor can be classified for arbitrary type parameters P, it can be classified for any realisation of these parameters: F is polymorphic in P. We discharge the assumption $[\mathrm{X} : \mathcal{S}]$, universally quantify over the type parameters, and add the assumption $[\mathrm{F} : \forall P.\mathcal{S} \to \mathcal{X}']$ to the context. Classifying the scope b of the functor definition yields the type of the entire phrase (the functor is only defined locally).

$$\frac{}{\mathcal{C} \vdash \epsilon_{\mathrm{b}} : \exists\emptyset.\epsilon_{\mathcal{S}}} \tag{T-18}$$

(T-18) The empty structure body introduces an empty quantifier.

Structure Expressions $\boxed{\mathcal{C} \vdash \mathrm{s} : \mathcal{X}}$

$$\frac{\mathcal{C} \vdash \mathrm{sp} : \mathcal{S}}{\mathcal{C} \vdash \mathrm{sp} : \exists\emptyset.\mathcal{S}} \tag{T-19}$$

(T-19) The classification of a path is a structure. Existentially quantifying over the empty set reflects the fact that a path cannot introduce new types.

$$\frac{\mathcal{C} \vdash \mathrm{b} : \mathcal{X}}{\mathcal{C} \vdash \textbf{struct}\ \mathrm{b}\ \textbf{end} : \mathcal{X}} \tag{T-20}$$

$$\frac{\begin{array}{ll} \mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S}' & \\ \mathcal{C}(\mathrm{F}) = \forall Q.\mathcal{S}'' \to \mathcal{X} & P \cap \mathrm{FV}(\forall Q.\mathcal{S}'' \to \mathcal{X}) = \emptyset \\ \mathcal{S}' \succeq \varphi(\mathcal{S}'') & \mathrm{Dom}(\varphi) = Q \\ \varphi(\mathcal{X}) = \exists P'.\mathcal{S} & P \cap P' = \emptyset \end{array}}{\mathcal{C} \vdash \mathrm{F}\ \mathrm{s} : \exists P \cup P'.\mathcal{S}} \tag{T-21}$$

(T-21) Assume the argument s has existential structure $\exists P.\mathcal{S}'$. We locally eliminate the quantifier to see whether the functor may be applied to the structure (a combination of realisation and enrichment), obtaining the existentially quantified functor result $\varphi(\mathcal{X}) = \exists P'.\mathcal{S}$. By virtue of the realisation, the functor may propagate some of the hypothetical types in P from the actual argument to the result. To prevent them escaping their scope, we extend the existential quantification over the actual result $\mathcal{S}$ to hide both P and P'. The side conditions on P ensure that these hypothetical types are not accidentally confused with existing types, nor with the types returned by the application. They can always be satisfied by suitable renamings of P and P'.

$$\frac{\begin{array}{ll}\mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S} & \\ \mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P'.\mathcal{S}' & P \cap \mathrm{FV}(\Lambda P'.\mathcal{S}') = \emptyset \\ \mathcal{S} \succeq \varphi(\mathcal{S}') & \mathrm{Dom}(\varphi) = P'\end{array}}{\mathcal{C} \vdash \mathrm{s} \succeq \mathrm{S} : \exists P.\varphi(\mathcal{S}')} \quad \text{(T-22)}$$

(T-22) The signature expression S denotes a family of semantic structures, $\Lambda P'.\mathcal{S}'$. The curtailment s $\succeq$ S checks whether the type of s is at least as rich as some member $\varphi(\mathcal{S}')$ of this family. Since s has existential type $\exists P.\mathcal{S}$, we must first eliminate the existential before checking enrichment, ensuring that P is not accidentally confused with the free type variables of $\Lambda P'.\mathcal{S}'$. Since φ is applied to $\mathcal{S}'$ in the result $\exists P.\varphi(\mathcal{S}')$, the actual identities of type components merely specified in S is retained: the visibility and generality of some components of s, however, may be curtailed. The realisation may mention variables in P. The existential quantification over P in the result prevents these hypothetical types from escaping their scope.

$$\frac{\begin{array}{ll}\mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S} & \\ \mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P'.\mathcal{S}' & P \cap \mathrm{FV}(\Lambda P'.\mathcal{S}') = \emptyset \\ \mathcal{S} \succeq \varphi(\mathcal{S}') & \mathrm{Dom}(\varphi) = P'\end{array}}{\mathcal{C} \vdash \mathrm{s} \setminus \mathrm{S} : \exists P'.\mathcal{S}'} \quad \text{(T-23)}$$

(T-23) As in Rule (T-22) we require that there be *some* realisation φ such that $\mathcal{S}$ matches the signature $\Lambda P'.\mathcal{S}'$. However, the type of s $\setminus$ S is $\exists P'.\mathcal{S}'$, not $\exists P.\varphi(\mathcal{S}')$. As a result, types merely specified in S are made abstract.

4.2.2 An Example

We can now revisit the example in Section 4.1.2, Figure 4.2(a), to see how classification using existential structures manages to distinguish between

$$\mathbf{structure}\ \lfloor_{(\mathbf{t}=\alpha,\mathbf{x}:\mathbf{int}\to\alpha,\mathbf{y}:\alpha\to\mathbf{int})}\mathbf{X} = \lceil^{\exists\{\alpha\}.(\mathbf{t}=\alpha,\mathbf{x}:\mathbf{int}\to\alpha,\mathbf{y}:\alpha\to\mathbf{int})}\mathbf{struct\ datatype\ t}_{\alpha}\ =\ \mathbf{int\ with\ x,y\ end};$$

$$\mathbf{structure}\ \lfloor_{(\mathbf{X}:(),\mathbf{u}=\beta,\mathbf{x}':(\mathbf{int}\to\mathbf{int})\to\beta,\mathbf{y}':\beta\to(\mathbf{int}\to\mathbf{int}))}\mathbf{Y} = \lceil^{\exists\{\alpha\}.(\mathbf{X}:(),\mathbf{u}=\alpha,\mathbf{x}':(\mathbf{int}\to\mathbf{int})\to\alpha,\mathbf{y}':\alpha\to(\mathbf{int}\to\mathbf{int}))}\mathbf{struct\ structure\ X}\ =\ \mathbf{struct\ \ end};$$

$$\mathbf{datatype\ u}_{\alpha}\ =\ \mathbf{int}\to\mathbf{int\ with\ x}',\mathbf{y}'$$

$$\mathbf{end};$$

$$\mathbf{val\ z} = \underline{(\mathbf{Y}.\mathbf{y}'_{\beta\to\mathbf{int}\to\mathbf{int}}(\mathbf{X}.\mathbf{x}_{\mathbf{int}\to\alpha}\ 1)_{\alpha}\ )}\ 2$$

(a) A partial, correctly unsuccessful classification of the phrase in Figure 4.2(a). The state-less classification using existential types manages to prevent the offending subphrase from being accepted. The type violation is underlined.

$$\mathbf{structure}\ \lfloor_{(\mathbf{t}=\alpha,\mathbf{x}:\mathbf{int}\to\alpha,\mathbf{y}:\alpha\to\mathbf{int})}\mathbf{X} = \lceil^{(\mathbf{t}=\alpha,\mathbf{x}:\mathbf{int}\to\alpha,\mathbf{y}:\alpha\to\mathbf{int})}\mathbf{struct\ datatype\ t}_{\alpha}\ =\ \mathbf{int\ with\ x,y\ end};$$

$$\mathbf{structure}\ \lfloor_{(\mathbf{X}:(),\mathbf{u}=\alpha,\mathbf{x}':(\mathbf{int}\to\mathbf{int})\to\alpha,\mathbf{y}':\alpha\to(\mathbf{int}\to\mathbf{int}))}\mathbf{Y} = \lceil^{(\mathbf{X}:(),\mathbf{u}=\alpha,\mathbf{x}':(\mathbf{int}\to\mathbf{int})\to\alpha,\mathbf{y}':\alpha\to(\mathbf{int}\to\mathbf{int}))}\mathbf{struct\ structure\ X}\ =\ \mathbf{struct\ \ end};$$

$$\mathbf{datatype\ u}_{\alpha}\ =\ \mathbf{int}\to\mathbf{int\ with\ x}',\mathbf{y}'$$

$$\mathbf{end};$$

$$\mathbf{val\ z} = (\mathbf{Y}.\mathbf{y}'_{\alpha\to\mathbf{int}\to\mathbf{int}}(\mathbf{X}.\mathbf{x}_{\mathbf{int}\to\alpha}\ 1)_{\alpha}\ )_{\mathbf{int}\to\mathbf{int}}\ 2$$

(b) The unsound classification of Figure 4.2(c) annotated with types.

Figure 4.6: The example in Figure 4.2(a) revisited.

abstract types that need to be kept distinct, without relying on the use of a global state of generated type variables. In Figure 4.6(a), we have indicated the semantic existential structures of the two key structure expressions using the notation $\overset{\mathcal{X}}{\lceil}$s, and the semantic structures, with which the identifiers **X** and **Y** are declared in the context, using the notation $\underset{\mathcal{S}}{\lfloor}$X. In addition, we've annotated the defining occurrences of **t** and **u** with the type variables chosen to represent them at their point of definition.

Let's assume the initial context is empty. The existential type of the structure expression defining **X** is:

$$\exists\{\alpha\}.(\mathbf{t} = \alpha, \mathbf{x} : \mathbf{int} \to \alpha, \mathbf{y} : \alpha \to \mathbf{int}).$$

Since α is fresh for the empty context, we can eliminate this existential quantifier directly so that, after the definition of **X**, the context of **Y** is:

$$[\mathbf{X} : (\mathbf{t} = \alpha, \mathbf{x} : \mathbf{int} \to \alpha, \mathbf{y} : \alpha \to \mathbf{int})],$$

containing a free occurrence of α. Still, as in the state-less classification of Figure 4.2(b), we are free to re-use α to represent **u** at the definition of **u**, since α no longer occurs in the context after the second definition of **X**. However, examining the existential structure,

$$\exists\{\alpha\}.(\mathbf{X} : (), \mathbf{u} = \alpha, \mathbf{x}' : (\mathbf{int} \to \mathbf{int}) \to \alpha, \mathbf{y}' : \alpha \to (\mathbf{int} \to \mathbf{int})),$$

of the structure expression defining **Y**, we can see that this variable is distinguished from the free occurrence of α in the context by the fact that it existentially bound. According to Rule (T-15), in order to extend the context with the definition of **Y**, we need to first eliminate this existential quantifier. The first side-condition of Rule (T-15) only permits us to do this in a way that avoids capturing the free occurrence of α in the context of **Y**. To do this, it suffices to choose a renaming,

$$\exists\{\beta\}.(\mathbf{X} : (), \mathbf{u} = \beta, \mathbf{x}' : (\mathbf{int} \to \mathbf{int}) \to \beta, \mathbf{y}' : \beta \to (\mathbf{int} \to \mathbf{int})),$$

of

$$\exists\{\alpha\}.(\mathbf{X} : (), \mathbf{u} = \alpha, \mathbf{x}' : (\mathbf{int} \to \mathbf{int}) \to \alpha, \mathbf{y}' : \alpha \to (\mathbf{int} \to \mathbf{int})),$$

for a variable β that is *locally* fresh for the context of **Y**, and, in particular, distinct from α. Then, eliminating the renamed quantifier and extending the context by the declaration:

$$[\mathbf{Y} : (\mathbf{X} : (), \mathbf{u} = \beta, \mathbf{x}' : (\mathbf{int} \to \mathbf{int}) \to \beta, \mathbf{y}' : \beta \to (\mathbf{int} \to \mathbf{int}))],$$

ensures that the abstract types $\mathbf{X}.\mathbf{t}$ and $\mathbf{Y}.\mathbf{u}$ are correctly distinguished by distinct variables α and β, resulting in the detection of the type violation in the definition of $\mathbf{z}$.

The way in which the putatively simpler, state-less semantics used in Figure 4.2(b) managed to get it wrong is summarised in Figure 4.6(b), that uses similar annotations to the ones used in Figure 4.6(a). Notice that the representation α of $\mathbf{u}$ is free in the type

$$(\mathbf{X} : (), \mathbf{u} = \alpha, \mathbf{x}' : (\mathbf{int} \to \mathbf{int}) \to \alpha, \mathbf{y}' : \alpha \to (\mathbf{int} \to \mathbf{int}))$$

of the structure expression defining $\mathbf{Y}$. This means that it cannot be distinguished from the free occurrence of α in the context of $\mathbf{Y}$. Even though α is locally fresh for the context of the definition of $\mathbf{u}$, choosing α to represent $\mathbf{u}$ is unsound, because it eventually escapes into the type $\mathbf{Y}$, without being fresh for the context of $\mathbf{Y}$.

4.3 The Equivalence of the Generative and State-less Classification Judgements

In this section, we prove the equivalence of the generative and state-less classification judgements. Before proceeding to the statement of the main result, we will need a few more concepts:

Definition 4.2 (Solvable Structures and Signatures). The predicates $\exists P \vdash \mathcal{S}$ **Slv** and $\vdash \mathcal{L}$ **Slv** are defined as the least relations closed under the rules in Figure 4.7.

Intuitively, $\vdash \Lambda P.\mathcal{S}$ **Slv** holds if, and only if, every type parameter $\alpha \in P$ of the signature first occurs in a type binding $\mathrm{t} = \alpha$ within $\mathcal{S}$. The semantic signatures that arise as the denotations of signature expressions are invariably solvable:

Lemma 4.3 (Solvability). *If $\mathcal{C} \vdash \mathrm{S} \rhd \mathcal{L}$ then $\vdash \mathcal{L}$* **Slv**.

Proof. *A simple induction on the rules defining $\mathcal{C} \vdash \mathrm{B} \rhd \mathcal{L}$ and $\mathcal{C} \vdash \mathrm{S} \rhd \mathcal{L}$.*

Intuitively, the solvability of a signature ensures that whenever a structure matches the signature, then the corresponding realisation is unique. Moreover, the region of this realisation will only mention type variables already free in the structure (the region of a realisation was defined in Definition 3.13 (Realisations)). The latter is captured by the following Lemma:

Solvable Structures $\boxed{\exists P \vdash \mathcal{S}\ \mathbf{Slv}}$

$$\frac{}{\exists\emptyset \vdash \epsilon_{\mathcal{S}}\ \mathbf{Slv}}$$

$$\frac{\mathrm{FV}(\tau) \cap P = \emptyset \quad \exists P \vdash \mathcal{S}\ \mathbf{Slv}}{\exists P \vdash \mathrm{t} = \tau, \mathcal{S}\ \mathbf{Slv}}$$

$$\frac{\alpha \notin P \quad \exists P \vdash \mathcal{S}\ \mathbf{Slv}}{\exists\{\alpha\} \cup P \vdash \mathrm{t} = \alpha, \mathcal{S}\ \mathbf{Slv}}$$

$$\frac{\mathrm{FV}(v) \cap P = \emptyset \quad \exists P \vdash \mathcal{S}\ \mathbf{Slv}}{\exists P \vdash \mathrm{x} : v, \mathcal{S}\ \mathbf{Slv}}$$

$$\frac{\exists P \vdash \mathcal{S}\ \mathbf{Slv} \quad \exists P' \vdash \mathcal{S}'\ \mathbf{Slv} \quad P \cap P' = \emptyset}{\exists P \cup P' \vdash \mathrm{X} : \mathcal{S}, \mathcal{S}'\ \mathbf{Slv}}$$

Solvable Signatures $\boxed{\vdash \mathcal{L}\ \mathbf{Slv}}$

$$\frac{\exists P \vdash \mathcal{S}\ \mathbf{Slv}}{\vdash \Lambda P.\mathcal{S}\ \mathbf{Slv}}$$

Figure 4.7: The definition of solvable structures and signatures.

Lemma 4.4 (Propagation).
If $\exists\mathrm{Dom}(\varphi) \vdash \mathcal{S}'\ \mathbf{Slv}$ *and* $\mathcal{S} \succeq \varphi(\mathcal{S}')$ *then* $\mathrm{Reg}(\varphi) \subseteq \mathrm{FV}(\mathcal{S})$.

Remark 4.3.1 *(Motivating Solvability).* We will not need the property that solvable signatures gives rise to unique matching realisations. However, it is worth mentioning that this is the key property one needs to prove that the classifications of structure bodies and expressions are unique. To see this, consider the Mini-SML signature $\mathcal{L}$ and structure $\mathcal{S}$ defined as:

$$\begin{array}{rcl} \mathcal{L} & \equiv & \Lambda\alpha.(\mathbf{x} : \alpha(\mathbf{int})), \\ \mathcal{S} & \equiv & (\mathbf{x} : \mathbf{int}). \end{array}$$

Observe that $\mathcal{L}$ fails to be solvable ($\not\vdash \mathcal{L}\ \mathbf{Slv}$) since its parameter α does not occur as the denotation of a type component within its body. Consider the two distinct realisations $\varphi_1 \equiv [\Lambda('b).'b/\alpha]$ and $\varphi_2 \equiv [\Lambda('b).\mathbf{int}/\alpha]$. It is easy to check that $\mathcal{S}$ matches $\mathcal{L}$ by either one, since in each case we have $\mathcal{S} \succeq \varphi_i\,(\mathbf{x} : \alpha(\mathbf{int}))$.

Definition 4.5 (Ground Functors and Contexts). The predicates $\vdash \mathcal{F}\ \mathbf{Gnd}$ and $\vdash \mathcal{C}\ \mathbf{Gnd}$ are defined as the least relations closed under the rules in Figure 4.8.

Ground Functors $\boxed{\vdash \mathcal{F}\ \textbf{Gnd}}$

$$\frac{\exists P \vdash \mathcal{S}\ \textbf{Slv}}{\vdash \forall P.\mathcal{S} \to \mathcal{X}\ \textbf{Gnd}}$$

Ground Contexts $\boxed{\vdash \mathcal{C}\ \textbf{Gnd}}$

$$\frac{\forall \mathrm{F} \in \mathrm{Dom}(\mathcal{C}).\vdash \mathcal{C}(\mathrm{F})\ \textbf{Gnd}}{\vdash \mathcal{C}\ \textbf{Gnd}}$$

Figure 4.8: The definition of ground functors and contexts.

Informally, a semantic functor is ground provided the signature of its argument is solvable; a context is ground provided all the functors in its domain are ground.

No matter which classification judgements we adopt, as long as a given semantic functor $\mathcal{F}$ is ground, whenever we apply a functor of this type, the free variables of the result are either propagated from the type of the actual argument, or were already free in $\mathcal{F}$:

Lemma 4.6 (Functor Propagation). *If* $\vdash \forall P.\mathcal{S} \to \mathcal{X}$ **Gnd** *and* $\mathcal{S}' \succeq \varphi(\mathcal{S})$*, where* $\mathrm{Dom}(\varphi) = P$*, then* $\mathrm{FV}(\varphi(\mathcal{X})) \subseteq \mathrm{FV}(\mathcal{S}') \cup \mathrm{FV}(\forall P.\mathcal{S} \to \mathcal{X})$*.*

Proof. *A simple consequence of Lemma 4.4 (Propagation).*

Remark 4.3.2 *(Motivating Groundedness).* Although we will not need this property, we should also point out that insisting on ground functors ensures that the type of a functor application is uniquely determined by the type of the actual argument. As a counter-example, consider the Mini-SML functor $\mathcal{F}$ defined as:

$$\mathcal{F} \quad \equiv \quad \forall \alpha.(\mathbf{x} : \alpha(\mathbf{int})) \to \exists \emptyset.(\mathbf{y} : \alpha(\mathbf{bool}))$$

Let $\mathcal{L}$, $\mathcal{S}$, φ_1 and φ_2 be defined as in Remark 4.3.1. Observe that $\mathcal{F}$ fails to be ground since its argument signature is the unsolvable signature $\mathcal{L}$. Consider the application F s, where F is a functor of type $\mathcal{F}$, and s is a structure expression of type $\mathcal{S}$. Recall that $\mathcal{S}$ matches $\mathcal{L}$ via both φ_1 and φ_2. Consequently, in either semantics, there are two ways of using the functor application rule, one for each choice of matching realisation. Choosing φ_1 yields the result type $\varphi_1(\mathbf{y} : \alpha(\mathbf{bool})) \equiv (\mathbf{y} : \mathbf{bool})$, with a boolean component. Choosing φ_2, on the other hand, yields the result type $\varphi_2(\mathbf{y} : \alpha(\mathbf{bool})) \equiv (\mathbf{y} : \mathbf{int})$, with an integer component. There is no principled way to select between these very different results. Insisting on ground functors excludes such examples.

We will need the following simple lemma:

Lemma 4.7 (Free Variables).

- $\mathcal{C} \vdash \mathrm{d} \triangleright d$ *implies* $\mathrm{FV}(d) \subseteq \mathrm{FV}(\mathcal{C})$.
- $\mathcal{C} \vdash \mathrm{v} \triangleright v$ *implies* $\mathrm{FV}(v) \subseteq \mathrm{FV}(\mathcal{C})$.
- $\mathcal{C} \vdash \mathrm{e} : v$ *implies* $\mathrm{FV}(v) \subseteq \mathrm{FV}(\mathcal{C})$.
- $\mathcal{C} \vdash \mathrm{sp} : \mathcal{S}$ *implies* $\mathrm{FV}(\mathcal{S}) \subseteq \mathrm{FV}(\mathcal{C})$.
- $\mathcal{C} \vdash \mathrm{do} \triangleright d$ *implies* $\mathrm{FV}(d) \subseteq \mathrm{FV}(\mathcal{C})$.
- $\mathcal{C} \vdash \mathrm{vo} : v$ *implies* $\mathrm{FV}(v) \subseteq \mathrm{FV}(\mathcal{C})$.
- $\mathcal{C} \vdash \mathrm{S} \triangleright \mathcal{L}$ *implies* $\mathrm{FV}(\mathcal{L}) \subseteq \mathrm{FV}(\mathcal{C})$.
- $\mathcal{C} \vdash \mathrm{B} \triangleright \mathcal{L}$ *implies* $\mathrm{FV}(\mathcal{L}) \subseteq \mathrm{FV}(\mathcal{C})$.

Proof (Sketch). *The proof follows easily by simultaneous induction on the rules defining the judgements. The first three clauses are Core language dependent, but must be proven together with the remaining statements in order deal with subphrases containing type and value occurrences.*

We will also need a similar lemma for the judgements $\mathcal{C} \vdash \mathrm{b} : \mathcal{X}$ and $\mathcal{C} \vdash \mathrm{s} : \mathcal{X}$. The lemma holds as long as the functors occurring in derivations are ground. This is only a technical restriction since we already discussed how the presence of non-ground functors leads to problems (cf. Remark 4.3.2). Indeed, by Lemma 4.3 (Solvability), we know that all signatures arising from signature expressions are solvable, so that functor definitions only introduce ground functors. However, because a context may contain arbitrary functor bindings, we need to impose a condition to ensure that any pre-declared functors are ground. This is the motivation for requiring that contexts are ground in the following lemma:

Lemma 4.8 (Free Variables).

- $\mathcal{C} \vdash \mathrm{s} : \mathcal{X}$ *implies* $\vdash \mathcal{C}$ **Gnd** *implies* $\mathrm{FV}(\mathcal{X}) \subseteq \mathrm{FV}(\mathcal{C})$*; and*
- $\mathcal{C} \vdash \mathrm{b} : \mathcal{X}$ *implies* $\vdash \mathcal{C}$ **Gnd** *implies* $\mathrm{FV}(\mathcal{X}) \subseteq \mathrm{FV}(\mathcal{C})$.

Proof (Sketch). *The proof follows easily by rule induction. The idea is to maintain the groundedness of the context as an invariant of the proof. In case* (T-17) *we appeal to Lemma 4.3 (Solvability) to ensure that the context extended with the functor binding is ground. In case* (T-21) *we appeal to Lemma 4.6 (Functor Propagation) to show that applying the matching realisation to the result does not introduce spurious type variables. Similarly, case T-22 requires an appeal to Lemmas 4.3 (Solvability) and 4.4 (Propagation). Case* (T-23) *follows by an appeal to Lemma 4.7 (Free Variables).*

In the previous chapter, we informally identified semantic objects that are equivalent up to capture-avoiding renamings of bound type variables. For the results in this chapter, we will need to make the identification more formal (though we will still refrain from spelling out all the details). We first define the concept of a renaming, which is similar to, but simpler than, the notion of realisation we already encountered.

Definition 4.9 (Renamings). A *renaming* is a kind-preserving, finite map from type variables to type variables. We let

$$\rho, \sigma, \pi \ \in \overset{\text{def}}{=} \{ f \in \mathit{TypVar} \overset{\text{fin}}{\to} \mathit{TypVar} \mid \forall \kappa . \forall \alpha^{\kappa} \in \mathrm{Dom}(f) . f(\alpha^{\kappa}) \in \mathit{TypVar}^{\kappa} \},$$

range over renamings.

We will use the more suggestive notation $[M/N]$ to denote a *bijective* renaming with domain N and range M that simply swaps variables. The effect of *applying* a renaming ρ to a variable α, written $\rho\langle\alpha\rangle$, is defined to be $\rho\langle\alpha\rangle \overset{\text{def}}{=}$ if $\alpha \in \mathrm{Dom}(\rho)$ then $\rho(\alpha)$ else α. We extend the operation of renaming free variables compositionally to all semantic objects in such a way that bound variables are renamed *only* when necessary to avoid capture (in the obvious way).

Let $\mathrm{Inv}(\rho) \overset{\text{def}}{=} \mathrm{Dom}(\rho) \cup \mathrm{Rng}(\rho)$ describe the set of variables *involved* in the renaming ρ.

We can now formally define the sense in which semantic objects are identified "up to renamings of bound variables":

Definition 4.10 (α-Equivalence).

- Two signatures $\mathcal{L} \equiv \Lambda P.\mathcal{S}$, $\mathcal{L}' \equiv \Lambda P'.\mathcal{S}'$, are α-equivalent, written $\mathcal{L} \overset{\alpha}{\equiv} \mathcal{L}'$, if, and only if, there is a *bijective* renaming $[P'/P]$ such that $[P'/P]\langle\mathcal{S}\rangle \equiv \mathcal{S}'$ and $\mathrm{FV}(\mathcal{L}) = \mathrm{FV}(\mathcal{L}')$.

- Two existential structures $\mathcal{X} \equiv \exists P.\mathcal{S}$, $\mathcal{X}' \equiv \exists P'.\mathcal{S}'$, are α-equivalent, written $\mathcal{X} \overset{\alpha}{\equiv} \mathcal{X}'$, if, and only if, there is a *bijective* renaming $[P'/P]$ such that $[P'/P]\langle\mathcal{S}\rangle \equiv \mathcal{S}'$ and $\mathrm{FV}(\mathcal{X}) = \mathrm{FV}(\mathcal{X}')$.
- Two functors $\mathcal{F} \equiv \forall P.\mathcal{S} \to \mathcal{X}$, $\mathcal{F}' \equiv \forall P'.\mathcal{S}' \to \mathcal{X}'$, are α-equivalent, written $\mathcal{F} \overset{\alpha}{\equiv} \mathcal{F}'$, if, and only if, there is a *bijective* renaming $[P'/P]$ such that $[P'/P]\langle\mathcal{S}\rangle \equiv \mathcal{S}'$, $[P'/P]\langle\mathcal{X}\rangle \overset{\alpha}{\equiv} \mathcal{X}'$ and $\mathrm{FV}(\mathcal{F}) = \mathrm{FV}(\mathcal{F}')$.

We identify all semantic objects that are α-equivalent.

Renamings enjoy the following properties:

Properties 4.11 (Renaming).

- *If* $\mathrm{Dom}(\rho) \cap \mathrm{FV}(\mathcal{O}) = \emptyset$ *then* $\rho\langle\mathcal{O}\rangle \equiv \mathcal{O}$.
- *If* $\mathrm{Dom}(\rho') \cap \mathrm{FV}(\mathcal{O}) = \emptyset$ *then* $(\rho + \rho')\langle\mathcal{O}\rangle \equiv \rho\langle\mathcal{O}\rangle$.
- *If* $\mathrm{Dom}(\rho) \cap \mathrm{Rng}(\rho') = \emptyset$ *then* $(\rho + \rho')\langle\mathcal{O}\rangle \equiv \rho\langle\rho'\langle\mathcal{O}\rangle\rangle$.
- *If* $(\rho \downarrow \mathrm{FV}(\mathcal{O})) = (\rho' \downarrow \mathrm{FV}(\mathcal{O}))$ *then* $\rho\langle\mathcal{O}\rangle \equiv \rho'\langle\mathcal{O}\rangle$.
- *If* $\mathrm{Inv}(\rho) \cap P = \emptyset$ *then*
  - $\rho\langle\Lambda P.\mathcal{S}\rangle \equiv \Lambda P.\rho\langle\mathcal{S}\rangle$,
  - $\rho\langle\exists P.\mathcal{S}\rangle \equiv \exists P.\rho\langle\mathcal{S}\rangle$, *and*
  - $\rho\langle\forall P.\mathcal{S} \to \mathcal{X}\rangle \equiv \forall P.\rho\langle\mathcal{S}\rangle \to \rho\langle\mathcal{X}\rangle$.

In our proofs, we shall frequently make implicit appeals to these properties.

We are now in a position to state the main result of this chapter:

Theorem 4.12 (Equivalence). *Provided* $\vdash \mathcal{C}$ **Gnd** *and* $\mathcal{C}, N$ **rigid***:*

(Completeness)

- *If* $\mathcal{C}, N \vdash \mathrm{b} : \mathcal{S} \Rightarrow P$ *then* $\mathcal{C} \vdash \mathrm{b} : \exists P.\mathcal{S}$.
- *If* $\mathcal{C}, N \vdash \mathrm{s} : \mathcal{S} \Rightarrow P$ *then* $\mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S}$.

(Soundness)

- *If* $\mathcal{C} \vdash \mathrm{b} : \mathcal{X}$ *then, for some* P *and* $\mathcal{S}$, $\mathcal{C}, N \vdash \mathrm{b} : \mathcal{S} \Rightarrow P$ *with* $\mathcal{X} \overset{\alpha}{\equiv} \exists P.\mathcal{S}$.

- *If* $\mathcal{C} \vdash \mathrm{s} : \mathcal{X}$ *then, for some* P *and* $\mathcal{S}$, $\mathcal{C}, N \vdash \mathrm{s} : \mathcal{S} \Rightarrow P$ *with* $\mathcal{X} \overset{\alpha}{\equiv} \exists P.\mathcal{S}$.

It might help to explain the provisos on Theorem 4.12. We already motivated the restriction to ground contexts, so let us consider the rigidity requirement. Informally, with the generative classification judgements, it is only if we start with a rigid context that the set of variables returned by the classification of a phrase will be distinct from the variables occurring free in the context. Violating this condition leads to unsound judgements. For this reason, the only generative judgements of actual interest to us are those that mention rigid contexts. Because it covers the cases that we are interested in, the fact that Theorem 4.12 only holds for judgements with respect to ground and rigid contexts is merely a technical restriction.

The next two sections are devoted to the proof of Theorem 4.12.

4.3.1 Completeness

An operational view of the state-less classification judgements is that we have replaced the notion of *global* generativity by *local* freshness, using the ability to rename existentially bound variables whenever necessary to avoid capture of free variables. The proof of completeness is easy because any variable that is globally generative with respect to both the state and the context, will also be locally fresh with respect to the context, enabling a straightforward construction of a corresponding state-less derivation.

Proof (Completeness). *We use strong rule induction on the generative classification judgements to prove the theorems:*

$$\mathcal{C}, N \vdash \mathrm{b} : \mathcal{S} \Rightarrow P \supset \vdash \mathcal{C}\ \mathbf{Gnd} \supset \mathcal{C}, N\ \mathbf{rigid} \supset \mathcal{C} \vdash \mathrm{b} : \exists P.\mathcal{S}$$

$$\mathcal{C}, N \vdash \mathrm{s} : \mathcal{S} \Rightarrow P \supset \vdash \mathcal{C}\ \mathbf{Gnd} \supset \mathcal{C}, N\ \mathbf{rigid} \supset \mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S}$$

We will only consider the case for structure definitions, the other cases are similar:

$\boxed{\textbf{E-15}}$ *By strong induction we may assume the premises:*

$$\mathcal{C}, N \vdash \mathrm{s} : \mathcal{S} \Rightarrow P, \tag{1}$$

$$\mathcal{C}[\mathrm{X} : \mathcal{S}], N \cup P \vdash \mathrm{b} : \mathcal{S}' \Rightarrow P', \tag{2}$$

$$\mathrm{X} \notin \mathrm{Dom}(\mathcal{S}'). \tag{3}$$

and the induction hypotheses:

$$\vdash \mathcal{C}\ \mathbf{Gnd} \supset \mathcal{C}, N\ \mathbf{rigid} \supset \mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S}, \tag{4}$$

$$\vdash \mathcal{C}[\mathrm{X} : \mathcal{S}]\ \mathbf{Gnd} \supset \mathcal{C}[\mathrm{X} : \mathcal{S}], N \cup P\ \mathbf{rigid} \supset \mathcal{C}[\mathrm{X} : \mathcal{S}] \vdash \mathrm{b} : \exists P'.\mathcal{S}'. \tag{5}$$

We need to show:

$$\vdash \mathcal{C}\ \mathbf{Gnd} \supset \mathcal{C}, N\ \mathbf{rigid} \supset \mathcal{C} \vdash \mathbf{structure}\ \mathrm{X}\ =\ \mathrm{s;b} : \exists P \cup P'.\mathrm{X} : \mathcal{S}, \mathcal{S}'. \tag{6}$$

Assume:

$$\vdash \mathcal{C}\ \mathbf{Gnd}, \tag{7}$$

$$\mathcal{C}, N\ \mathbf{rigid}. \tag{8}$$

By induction hypothesis (4) *on* (7) *and* (8) *we obtain:*

$$\mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S}. \tag{9}$$

Property 3.31 (Generativity) of (1)*, together with* (8)*, ensures that:*

$$P \cap \mathrm{FV}(\mathcal{C}) = \emptyset. \tag{10}$$

Since we are only declaring a structure (not a functor), (7) *extends to:*

$$\vdash \mathcal{C}[\mathbf{X} : \mathcal{S}]\ \mathbf{Gnd}. \tag{11}$$

Lemma 4.8 (Free Variables) on (7) *and* (9) *guarantees* $\mathrm{FV}(\exists P.\mathcal{S}) \subseteq \mathrm{FV}(\mathcal{C})$*. It follows from* (8) *that:*

$$\mathrm{FV}(\mathcal{S}) \subseteq N \cup P \tag{12}$$

and consequently:

$$\mathcal{C}[\mathbf{X} : \mathcal{S}], N \cup P\ \mathbf{rigid}. \tag{13}$$

Induction hypothesis (5) *applied to* (11) *and* (13) *yields:*

$$\mathcal{C}[\mathrm{X} : \mathcal{S}] \vdash \mathrm{b} : \exists P'.\mathcal{S}'. \tag{14}$$

Property 3.31 (Generativity) of (2) *ensures* $P' \cap (N \cup P) = \emptyset$*, which, together with* (12)*, entails:*

$$P' \cap (P \cup \mathrm{FV}(\mathcal{S})) = \emptyset. \tag{15}$$

Rule (T-15) *on* (9)*,* (10)*,* (14) (15) *and* (3) *derives:*

$$\mathcal{C} \vdash \textbf{structure}\ \mathrm{X}\ =\ \mathrm{s;b} : \exists P \cup P'.\mathrm{X} : \mathcal{S}, \mathcal{S}'$$

as desired.

In the complete proof, Property 3.31 (Generativity) and Lemma 4.8 (Free Variables) conspire to ensure that the side conditions on bound variables, that are imposed by Rules (T-13) through (T-23) to prevent the capture of free variables, are immediately satisfied by the semantic objects classifying phrases in subderivations: implicit appeals to α-conversion are never required.

4.3.2 Soundness

Soundness is more difficult to prove, because the state-less classification judgements merely require subderivations to hold for *particular* choices of locally fresh variables. A variable may be locally fresh without being globally generative with respect to a given state. This foils naive attempts to directly construct a generative derivation from a state-less derivation.

To address this problem, we introduce a modified formulation of the classification judgements with the judgement forms $\mathcal{C} \vdash' \mathrm{b} : \mathcal{X}$ and $\mathcal{C} \vdash' \mathrm{s} : \mathcal{X}$ (note the prime on the $\vdash$). The modified rules appear in Figure 4.9. The rules for the other constructs remain the same (modulo replacing occurrences of judgements $\mathcal{C} \vdash \mathrm{b} : \mathcal{X}$ and $\mathcal{C} \vdash \mathrm{s} : \mathcal{X}$ by $\mathcal{C} \vdash' \mathrm{b} : \mathcal{X}$ and $\mathcal{C} \vdash' \mathrm{s} : \mathcal{X}$, respectively). Instead of requiring subderivations to hold for *particular* choices of fresh variables, the modified rules require them to hold for *every* renaming of these variables. This makes it easy to construct a generative derivation from the derivation of a generalised judgement. Note that the inference rules are no longer finitely branching, but the relations remain well-founded, admitting inductive arguments. This technique of introducing a generalised judgement is adapted from McKinna and Pollack's formalisation of α-conversion [MP93].

Structure Bodies $\boxed{\mathcal{C} \vdash' \mathrm{b} : \mathcal{X}}$

$$\frac{\begin{array}{ll}\mathcal{C} \vdash' \mathrm{s} : \exists P.\mathcal{S} & Q \cap (P \cup \mathrm{FV}(\mathcal{S})) = \emptyset \\ \forall \pi.\mathrm{Dom}(\pi) = P \supset \mathcal{C}[\mathrm{X} : \pi\langle\mathcal{S}\rangle] \vdash' \mathrm{b} : \pi\langle\exists Q.\mathcal{S}'\rangle & \mathrm{X} \notin \mathrm{Dom}(\mathcal{S}')\end{array}}{\mathcal{C} \vdash' \textbf{structure}\ \mathrm{X}\ =\ \mathrm{s;b} : \exists P \cup Q.\mathrm{X} : \mathcal{S}, \mathcal{S}'} \quad \text{(T'-15)}$$

$$\frac{\begin{array}{l}\mathcal{C} \vdash' \mathrm{s} : \exists P.\mathcal{S} \quad Q \cap P = \emptyset \\ \forall \pi.\mathrm{Dom}(\pi) = P \supset \mathcal{C}[\mathrm{X} : \pi\langle\mathcal{S}\rangle] \vdash' \mathrm{b} : \pi\langle\exists Q.\mathcal{S}'\rangle\end{array}}{\mathcal{C} \vdash' \textbf{local}\ \mathrm{X}\ =\ \mathrm{s}\ \textbf{in}\ \mathrm{b} : \exists P \cup Q.\mathcal{S}'} \quad \text{(T'-16)}$$

$$\frac{\begin{array}{l}\mathcal{C} \vdash \mathrm{S} \rhd \Lambda P.\mathcal{S} \\ \forall \pi.\mathrm{Dom}(\pi) = P \supset \mathcal{C}[\mathrm{X} : \pi\langle\mathcal{S}\rangle] \vdash' \mathrm{s} : \pi\langle\mathcal{X}'\rangle \\ \mathcal{C}[\mathrm{F} : \forall P.\mathcal{S} \to \mathcal{X}'] \vdash' \mathrm{b} : \mathcal{X}\end{array}}{\mathcal{C} \vdash' \textbf{functor}\ \mathrm{F}\ (\mathrm{X} : \mathrm{S})\ =\ \mathrm{s}\ \textbf{in}\ \mathrm{b} : \mathcal{X}} \quad \text{(T'-17)}$$

Figure 4.9: State-less Classification Judgements with Generalised Premises (modified rules only)

Our strategy for proving soundness is to first show that any derivation in the original system gives rise to a corresponding derivation in the generalised system:

Lemma 4.13 (Soundness — Part I).

- *If* $\mathcal{C} \vdash \mathrm{b} : \mathcal{X}$ *then* $\mathcal{C} \vdash' \mathrm{b} : \mathcal{X}$*.*
- *If* $\mathcal{C} \vdash \mathrm{s} : \mathcal{X}$ *then* $\mathcal{C} \vdash' \mathrm{s} : \mathcal{X}$*.*

This corresponds to proving a stronger induction principle for our classification judgements. We then prove that any derivation in the generalised system gives rise to a corresponding generative classification.

Lemma 4.14 (Soundness — Part II).
Provided $\vdash \mathcal{C}$ **Gnd** *and* $\mathcal{C}, N$ **rigid***,*

- *if* $\mathcal{C} \vdash' \mathrm{b} : \mathcal{X}$ *then we can find an* M *and* $\mathcal{S}$ *such that* $\mathcal{C}, N \vdash \mathrm{b} : \mathcal{S} \Rightarrow M$*, with* $\mathcal{X} \overset{\alpha}{\equiv} \exists M.\mathcal{S}$*.*
- *if* $\mathcal{C} \vdash' \mathrm{s} : \mathcal{X}$ *we can find an* M *and* $\mathcal{S}$ *such that* $\mathcal{C}, N \vdash \mathrm{s} : \mathcal{S} \Rightarrow M$*, with* $\mathcal{X} \overset{\alpha}{\equiv} \exists M.\mathcal{S}$*.*

Proof (Soundness). *Follows easily from Lemmas 4.13 and 4.14.*

Proof (Lemma 4.13). *We use rule induction on the classification rules to prove the stronger statements:*

$$\mathcal{C} \vdash \mathrm{b} : \mathcal{X} \supset \forall\rho.\rho\langle\mathcal{C}\rangle \vdash' \mathrm{b} : \rho\langle\mathcal{X}\rangle$$
$$\mathcal{C} \vdash \mathrm{s} : \mathcal{X} \supset \forall\rho.\rho\langle\mathcal{C}\rangle \vdash' \mathrm{s} : \rho\langle\mathcal{X}\rangle$$

Lemma 4.13 follows immediately by choosing ρ to be the empty (identity) renaming.

We will only consider the case of a structure definition. The other cases are similar.

$\boxed{\textbf{T-15}}$ *By induction we may assume:*

$$\forall\rho.\rho\langle\mathcal{C}\rangle \vdash' \mathrm{s} : \rho\langle\exists P.\mathcal{S}\rangle, \tag{1}$$

$$P \cap \mathrm{FV}(\mathcal{C}) = \emptyset, \tag{2}$$

$$\forall\rho.\rho\langle\mathcal{C}[\mathrm{X} : \mathcal{S}]\rangle \vdash' \mathrm{b} : \rho\langle\exists Q.\mathcal{S}'\rangle, \tag{3}$$

$$Q \cap (P \cup \mathrm{FV}(\mathcal{S})) = \emptyset, \tag{4}$$

$$\mathrm{X} \notin \mathrm{Dom}(\mathcal{S}'). \tag{5}$$

We need to show:

$$\forall\rho.\rho\langle\mathcal{C}\rangle \vdash' \textbf{structure}\ \mathrm{X}\ =\ \mathrm{s;b} : \rho\langle\exists P \cup Q.\mathrm{X} : \mathcal{S}, \mathcal{S}'\rangle.$$

Consider an arbitrary renaming ρ.

We first choose a bijective renaming $[\bar{P}/P]$ such that:

$$\bar{P} \cap (\mathrm{Inv}(\rho) \cup \mathrm{FV}(\exists P \cup Q.\mathrm{X} : \mathcal{S}, \mathcal{S}')) = \emptyset. \tag{6}$$

Then it is easy to verify that:

$$\exists P.\mathcal{S} \overset{\alpha}{\equiv} \exists\bar{P}.[\bar{P}/P]\langle\mathcal{S}\rangle. \tag{7}$$

Moreover, from (7) *and our choice of* $\bar{P}$*, we can show:*

$$\rho\langle\exists P.\mathcal{S}\rangle \overset{\alpha}{\equiv} \exists\bar{P}.\rho\langle[\bar{P}/P]\langle\mathcal{S}\rangle\rangle. \tag{8}$$

By induction hypothesis (1) *applied to* ρ *we have:*

$$\rho\langle\mathcal{C}\rangle \vdash' \mathrm{s} : \rho\langle\exists P.\mathcal{S}\rangle,$$

which by α*-equivalence* (8) *is also a derivation of:*

$$\rho\langle\mathcal{C}\rangle \vdash' \mathrm{s} : \exists\bar{P}.\rho\langle[\bar{P}/P]\langle\mathcal{S}\rangle\rangle. \tag{9}$$

We now choose a bijective renaming $[\bar{Q}/Q]$ *such that:*

$$\bar{Q} \cap (\mathrm{Inv}(\rho) \cup P \cup \bar{P} \cup \mathrm{FV}(\exists P \cup Q.\mathrm{X} : \mathcal{S}, \mathcal{S}')) = \emptyset. \tag{10}$$

Then it is easy to verify that:

$$\exists Q.\mathcal{S}' \overset{\alpha}{\equiv} \exists\bar{Q}.[\bar{Q}/Q]\langle\mathcal{S}'\rangle. \tag{11}$$

Moreover, from (11) *and our choice of* $\bar{Q}$*, it follows that:*

$$\rho\langle[\bar{P}/P]\langle\exists Q.\mathcal{S}'\rangle\rangle \overset{\alpha}{\equiv} \exists\bar{Q}.\rho\langle[\bar{P}/P]\langle[\bar{Q}/Q]\langle\mathcal{S}'\rangle\rangle\rangle. \tag{12}$$

Our choice of $\bar{Q}$ *also ensures:*

$$\bar{Q} \cap (\bar{P} \cup \mathrm{FV}(\rho\langle[\bar{P}/P]\langle\mathcal{S}\rangle\rangle)) = \emptyset. \tag{13}$$

We will now show:

$$\begin{aligned}&\forall\pi.\mathrm{Dom}(\pi) = \bar{P} \supset \\ &\quad \rho\langle\mathcal{C}\rangle[\mathrm{X} : \pi\langle\rho\langle[\bar{P}/P]\langle\mathcal{S}\rangle\rangle\rangle] \vdash' \mathrm{b} : \pi\langle\exists\bar{Q}.\rho\langle[\bar{P}/P]\langle[\bar{Q}/Q]\langle\mathcal{S}'\rangle\rangle\rangle\rangle.\end{aligned} \tag{14}$$

Consider an arbitrary renaming π *with* $\mathrm{Dom}(\pi) = \bar{P}$*.*
Define σ *to be the renaming* $\sigma \overset{\text{def}}{=} \rho + (\pi \circ [\bar{P}/P])$*.*
By induction hypothesis (3) *on the renaming* σ *we obtain:*

$$\sigma\langle\mathcal{C}[\mathrm{X} : \mathcal{S}]\rangle \vdash' \mathrm{b} : \sigma\langle\exists Q.\mathcal{S}'\rangle. \tag{15}$$

By reasoning about renamings we can prove equivalences (16)*,* (17) *and* (18) *below:*

$$\begin{aligned}\sigma\langle\mathcal{C}\rangle &= (\rho + (\pi \circ [\bar{P}/P]))\langle\mathcal{C}\rangle && \textit{by the definition of } \sigma \\ &= \rho\langle\mathcal{C}\rangle && \textit{since } P \cap \mathrm{FV}(\mathcal{C}) = \emptyset \end{aligned} \tag{16}$$

$$\begin{aligned}\sigma\langle\mathcal{S}\rangle &= (\rho + (\pi \circ [\bar{P}/P]))\langle\mathcal{S}\rangle && \textit{by the definition of } \sigma \\ &= \pi\langle\rho + [\bar{P}/P]\langle\mathcal{S}\rangle\rangle \\ & \qquad \textit{since } \mathrm{Dom}(\pi) \cap (\mathrm{Rng}(\rho) \cup (\mathrm{FV}(\mathcal{S}) \setminus P)) = \emptyset \\ &= \pi\langle\rho\langle[\bar{P}/P]\langle\mathcal{S}\rangle\rangle\rangle && \textit{since } \bar{P} \cap \mathrm{Dom}(\rho) = \emptyset \end{aligned} \tag{17}$$

$$\begin{aligned}\sigma\langle\exists Q.\mathcal{S}'\rangle &= (\rho + (\pi \circ [\bar{P}/P]))\langle\exists Q.\mathcal{S}'\rangle && \textit{by the definition of } \sigma \\ &= \pi\langle\rho + [\bar{P}/P]\langle\exists Q.\mathcal{S}'\rangle\rangle \\ & \qquad \textit{since } \mathrm{Dom}(\pi) \cap (\mathrm{Rng}(\rho) \cup (\mathrm{FV}(\exists Q.\mathcal{S}') \setminus P)) = \emptyset \\ &= \pi\langle\rho\langle[\bar{P}/P]\langle\exists Q.\mathcal{S}'\rangle\rangle\rangle && \textit{since } \bar{P} \cap \mathrm{Dom}(\rho) = \emptyset \\ &\overset{\alpha}{\equiv} \pi\langle\exists\bar{Q}.\rho\langle[\bar{P}/P]\langle[\bar{Q}/Q]\langle\mathcal{S}'\rangle\rangle\rangle\rangle && \textit{by } (12) \end{aligned} \tag{18}$$

Using equations (16), (17), and α-equivalence (18) we can re-express (15) as:

$$\rho\langle\mathcal{C}\rangle[\mathrm{X} : \pi\langle\rho\langle[\bar{P}/P]\langle\mathcal{S}\rangle\rangle\rangle] \vdash' \mathrm{b} : \pi\langle\exists\bar{Q}.\rho\langle[\bar{P}/P]\langle[\bar{Q}/Q]\langle\mathcal{S}'\rangle\rangle\rangle\rangle. \tag{19}$$

Since π was arbitrary we have established (14).

Clearly

$$\mathrm{X} \not\in \mathrm{Dom}(\rho\langle[\bar{P}/P]\langle[\bar{Q}/Q]\langle\mathcal{S}'\rangle\rangle\rangle) \tag{20}$$

follows from (5) since renaming the structure $\mathcal{S}'$ does not affect its domain.

Rule (T'-15) applied to (9), (13), (14) and (20) derives:

$$\begin{aligned}&\rho\langle\mathcal{C}\rangle \vdash' \textbf{structure}\ \mathrm{X}\ =\ \mathrm{s;b} : \\ &\quad \exists\bar{P} \cup \bar{Q}.\mathrm{X} : \rho\langle[\bar{P}/P]\langle\mathcal{S}\rangle\rangle, \rho\langle[\bar{P}/P]\langle[\bar{Q}/Q]\langle\mathcal{S}'\rangle\rangle\rangle.\end{aligned} \tag{21}$$

From our choice of $\bar{P}$ *and* $\bar{Q}$ *in* (6) *and* (10) *it is easy to verify that:*

$$\exists P \cup Q.\mathrm{X} : \mathcal{S}, \mathcal{S}' \overset{\alpha}{\equiv} \exists \bar{P} \cup \bar{Q}.\mathrm{X} : [\bar{P}/P]\langle\mathcal{S}\rangle, [\bar{P}/P]\langle[\bar{Q}/Q]\langle\mathcal{S}'\rangle\rangle. \quad (22)$$

Moreover, we have:

$$\begin{aligned} &\exists \bar{P} \cup \bar{Q}.\mathrm{X} : \rho\langle[\bar{P}/P]\langle\mathcal{S}\rangle\rangle, \rho\langle[\bar{P}/P]\langle[\bar{Q}/Q]\langle\mathcal{S}'\rangle\rangle\rangle \\ &= \rho\langle\exists \bar{P} \cup \bar{Q}.\mathrm{X} : [\bar{P}/P]\langle\mathcal{S}\rangle, [\bar{P}/P]\langle[\bar{Q}/Q]\langle\mathcal{S}'\rangle\rangle\rangle && (23) \\ &&& \textit{since } (\bar{P} \cup \bar{Q}) \cap \mathrm{Inv}(\rho) = \emptyset \textit{ by } (6) \textit{ and } (10) \\ &\overset{\alpha}{\equiv} \rho\langle\exists P \cup Q.\mathrm{X} : \mathcal{S}, \mathcal{S}'\rangle && \textit{by } (22) \quad (24) \end{aligned}$$

Using α*-equivalence* (24) *on judgement* (21) *yields:*

$$\rho\langle\mathcal{C}\rangle \vdash' \textbf{structure } \mathrm{X} \ = \ \mathrm{s;b} : \rho\langle\exists P \cup Q.\mathrm{X} : \mathcal{S}, \mathcal{S}'\rangle,$$

as desired.

Before proceeding with the proof of Lemma 4.14 we will require the counterpart to Lemma 4.8 (proof omitted but easy):

Lemma 4.15 (Free Variables).

- *If* $\vdash \mathcal{C}$ **Gnd** *and* $\mathcal{C} \vdash' \mathrm{b} : \mathcal{X}$ *then* $\mathrm{FV}(\mathcal{X}) \subseteq \mathrm{FV}(\mathcal{C})$.
- *If* $\vdash \mathcal{C}$ **Gnd** *and* $\mathcal{C} \vdash' \mathrm{s} : \mathcal{X}$ *then* $\mathrm{FV}(\mathcal{X}) \subseteq \mathrm{FV}(\mathcal{C})$.

Proof (Lemma 4.14). *We use strong rule induction on the generalised classification rules to prove the statements:*

$$\begin{aligned} &\mathcal{C} \vdash' \mathrm{b} : \mathcal{X} \supset \\ &\quad \vdash \mathcal{C} \textbf{ Gnd} \supset \\ &\quad\quad \forall N.\mathcal{C}, N \textbf{ rigid} \supset \\ &\quad\quad\quad \exists P, \mathcal{S}. \quad \mathcal{C}, N \vdash \mathrm{b} : \mathcal{S} \Rightarrow P \\ &\quad\quad\quad\quad\quad \wedge \mathcal{X} \overset{\alpha}{\equiv} \exists P.\mathcal{S} \end{aligned}$$

$$\begin{aligned} &\mathcal{C} \vdash' \mathrm{s} : \mathcal{X} \supset \\ &\quad \vdash \mathcal{C} \textbf{ Gnd} \supset \\ &\quad\quad \forall N.\mathcal{C}, N \textbf{ rigid} \supset \\ &\quad\quad\quad \exists P, \mathcal{S}. \quad \mathcal{C}, N \vdash \mathrm{s} : \mathcal{S} \Rightarrow P \\ &\quad\quad\quad\quad\quad \wedge \mathcal{X} \overset{\alpha}{\equiv} \exists P.\mathcal{S} \end{aligned}$$

We will only consider the case of a structure definition. The other cases are similar.

$\boxed{\textbf{T'-15}}$ *By strong induction we may assume the premises:*

$$\mathcal{C} \vdash' \mathrm{s} : \exists P.\mathcal{S}, \tag{1}$$

$$Q \cap (P \cup \mathrm{FV}(\mathcal{S})) = \emptyset, \tag{2}$$

$$\forall \pi.\mathrm{Dom}(\pi) = P \supset \mathcal{C}[\mathrm{X} : \pi\langle\mathcal{S}\rangle] \vdash' \mathrm{b} : \pi\langle\exists Q.\mathcal{S}'\rangle, \tag{3}$$

$$\mathrm{X} \notin \mathrm{Dom}(\mathcal{S}'), \tag{4}$$

and induction hypotheses:

$$\begin{array}{l} \vdash \mathcal{C}\ \mathbf{Gnd} \supset \\ \quad \forall N.\mathcal{C}, N\ \mathbf{rigid} \supset \\ \quad\quad \exists \bar{P}, \bar{\mathcal{S}}. \quad \mathcal{C}, N \vdash \mathrm{s} : \bar{\mathcal{S}} \Rightarrow \bar{P} \\ \quad\quad\quad \wedge \exists P.\mathcal{S} \overset{\alpha}{\equiv} \exists \bar{P}.\bar{\mathcal{S}} \end{array} \tag{5}$$

$$\forall \pi.\mathrm{Dom}(\pi) = P \supset \begin{array}{l} \vdash \mathcal{C}[\mathrm{X} : \pi\langle\mathcal{S}\rangle]\ \mathbf{Gnd} \supset \\ \quad \forall N.\mathcal{C}[\mathrm{X} : \pi\langle\mathcal{S}\rangle], N\ \mathbf{rigid} \supset \\ \quad\quad \exists \bar{Q}, \bar{\mathcal{S}}'. \quad \mathcal{C}[\mathrm{X} : \pi\langle\mathcal{S}\rangle], N \vdash \mathrm{b} : \bar{\mathcal{S}}' \Rightarrow \bar{Q} \\ \quad\quad\quad \wedge \pi\langle\exists Q.\mathcal{S}'\rangle \overset{\alpha}{\equiv} \exists \bar{Q}.\bar{\mathcal{S}}' \end{array} \tag{6}$$

We need to show:

$$\begin{array}{l} \vdash \mathcal{C}\ \mathbf{Gnd} \supset \\ \quad \forall N.\mathcal{C}, N\ \mathbf{rigid} \supset \\ \quad\quad \exists \hat{P}, \hat{\mathcal{S}}. \quad \mathcal{C}, N \vdash \mathbf{structure}\ \mathrm{X}\ =\ \mathrm{s;b} : \hat{\mathcal{S}} \Rightarrow \hat{P} \\ \quad\quad\quad \wedge \exists P \cup Q.\mathrm{X} : \mathcal{S}, \mathcal{S}' \overset{\alpha}{\equiv} \exists \hat{P}.\hat{\mathcal{S}} \end{array}$$

Assume:

$$\vdash \mathcal{C}\ \mathbf{Gnd}. \tag{7}$$

Consider an arbitrary N such that:

$$\mathcal{C}, N\ \mathbf{rigid}. \tag{8}$$

By induction hypothesis (5) *applied to* (7) *and* (8) *we obtain:*

$$\mathcal{C}, N \vdash \mathrm{s} : \bar{\mathcal{S}} \Rightarrow \bar{P}. \tag{9}$$

for some $\bar{P}$, $\bar{\mathcal{S}}$ satisfying:

$$\exists P.\mathcal{S} \overset{\alpha}{\equiv} \exists \bar{P}.\bar{\mathcal{S}}. \tag{10}$$

By (10) we must have some bijective renaming $[\bar{P}/P]$ such that:

$$[\bar{P}/P]\langle\mathcal{S}\rangle = \bar{\mathcal{S}}. \tag{11}$$

Since we are only declaring a structure (not a functor), (7) extends to:

$$\vdash \mathcal{C}[\mathrm{X} : [\bar{P}/P]\langle\mathcal{S}\rangle]\ \mathbf{Gnd}. \tag{12}$$

Lemma 4.15 (Free Variables) on (1) and (7) ensures:

$$\mathrm{FV}(\exists P.\mathcal{S}) \subseteq \mathrm{FV}(\mathcal{C}). \tag{13}$$

Combining (13) with (8), (10) and (11) we can establish:

$$\mathrm{FV}([\bar{P}/P]\langle\mathcal{S}\rangle) \subseteq N \cup \bar{P}. \tag{14}$$

Hence we can extend (8) to:

$$\mathcal{C}[\mathrm{X} : [\bar{P}/P]\langle\mathcal{S}\rangle], N \cup \bar{P}\ \mathbf{rigid}. \tag{15}$$

Applying induction hypothesis (6) to the renaming $[\bar{P}/P]$, using (12), $N \cup \bar{P}$, (15) and equation (11) we obtain:

$$\mathcal{C}[\mathrm{X} : \bar{\mathcal{S}}], N \cup \bar{P} \vdash \mathrm{b} : \bar{\mathcal{S}}' \Rightarrow \bar{Q} \tag{16}$$

for some $\bar{Q}$, $\bar{\mathcal{S}}'$ satisfying:

$$[\bar{P}/P]\langle\exists Q.\mathcal{S}'\rangle \overset{\alpha}{\equiv} \exists \bar{Q}.\bar{\mathcal{S}}'. \tag{17}$$

Furthermore, as a consequence of (4) and (17) we must also have:

$$\mathrm{X} \notin \mathrm{Dom}(\bar{\mathcal{S}}'). \tag{18}$$

Rule (E-15) applied to (9), (16) and (18) derives:

$$\mathcal{C}, N \vdash \mathbf{structure}\ \mathrm{X} \ = \ \mathrm{s;b} : \mathrm{X} : \bar{\mathcal{S}}, \bar{\mathcal{S}}' \Rightarrow \bar{P} \cup \bar{Q}. \tag{19}$$

It remains to show:

$$\exists P \cup Q.\mathrm{X} : \mathcal{S}, \mathcal{S}' \overset{\alpha}{\equiv} \exists \bar{P} \cup \bar{Q}.\mathrm{X} : \bar{\mathcal{S}}, \bar{\mathcal{S}}'.$$

Assembling the premises (1), (2), (3) *and* (4) *and applying Rule* (T'-15) *we obtain the original derivation of:*

$$\mathcal{C} \vdash' \textbf{structure}\ \mathrm{X}\ =\ \mathrm{s;b} : \exists P \cup Q.\mathrm{X} : \mathcal{S}, \mathcal{S}'. \tag{20}$$

Lemma 4.15 (Free Variables) on (20) *using* (7) *ensures:*

$$\mathrm{FV}(\exists P \cup Q.\mathrm{X} : \mathcal{S}, \mathcal{S}') \subseteq \mathrm{FV}(\mathcal{C}). \tag{21}$$

First, observe that, as a consequence of Property 3.31 (Generativity) applied to both (9) *and* (16), *we have:*

$$N \cap \bar{P} = \emptyset, \tag{22}$$

and

$$(N \cup \bar{P}) \cap \bar{Q} = \emptyset. \tag{23}$$

By Definition 3.32 (Rigidity) on (8), *combined with* (21), (22) *and* (23) *we have:*

$$\bar{P} \cap \mathrm{FV}(\exists P \cup Q.\mathrm{X} : \mathcal{S}, \mathcal{S}') = \emptyset, \tag{24}$$

$$\bar{Q} \cap \mathrm{FV}(\exists P \cup Q.\mathrm{X} : \mathcal{S}, \mathcal{S}') = \emptyset, \tag{25}$$

and:

$$\bar{P} \cap \bar{Q} = \emptyset. \tag{26}$$

Now choose a bijective renaming $[\hat{Q}/Q]$ *such that:*

$$\hat{Q} \cap (P \cup \bar{P} \cup Q \cup \bar{Q} \cup \mathrm{FV}(\exists P \cup Q.\mathrm{X} : \mathcal{S}, \mathcal{S}')) = \emptyset. \tag{27}$$

By our choice of $[\hat{Q}/Q]$ *we also have:*

$$\hat{Q} \cap \mathrm{FV}(\exists Q.\mathcal{S}') = \emptyset. \tag{28}$$

Hence it is easy to verify that:

$$\exists Q.\mathcal{S}' \overset{\alpha}{\equiv} \exists \hat{Q}.[\hat{Q}/Q]\langle \mathcal{S}' \rangle. \tag{29}$$

We can now show:

$$\begin{array}{lll}
& \exists \bar{Q}.\bar{\mathcal{S}}' & \\
\overset{\alpha}{\equiv} & [\bar{P}/P]\langle \exists Q.\mathcal{S}' \rangle & \textit{(by (17))} \\
\overset{\alpha}{\equiv} & [\bar{P}/P]\langle \exists \hat{Q}.[\hat{Q}/Q]\langle \mathcal{S}' \rangle \rangle & \textit{(by (29))} \\
= & \exists \hat{Q}.[\bar{P}/P]\langle [\hat{Q}/Q]\langle \mathcal{S}' \rangle \rangle & \textit{(since } \hat{Q} \cap \mathrm{Inv}([\bar{P}/P]) = \emptyset \textit{ by (27))}
\end{array} \tag{30}$$

Hence there is some bijection $[\bar{Q}/\hat{Q}]$ *such that:*

$$\bar{\mathcal{S}}' = [\bar{Q}/\hat{Q}]\langle[\bar{P}/P]\langle[\hat{Q}/Q]\langle\mathcal{S}'\rangle\rangle\rangle \tag{31}$$

With these observations it is easy to show:

$\exists P \cup Q.\mathrm{X} : \mathcal{S}, \mathcal{S}'$

$\overset{\alpha}{\equiv} \exists P \cup \hat{Q}.\mathrm{X} : [\hat{Q}/Q]\langle\mathcal{S}\rangle, [\hat{Q}/Q]\langle\mathcal{S}'\rangle$

(since $[\hat{Q}/Q]$ *bijective, and*

$\hat{Q} \cap (P \cup \mathrm{FV}(\exists P \cup Q.\mathrm{X} : \mathcal{S}, \mathcal{S}')) = \emptyset$

follows from (27)*)*

$$\overset{\alpha}{\equiv} \exists P \cup \hat{Q}.\mathrm{X} : \mathcal{S}, [\hat{Q}/Q]\langle\mathcal{S}'\rangle \tag{32}$$

(since $Q \cap \mathrm{FV}(\mathcal{S}) = \emptyset$ *by* (2)*)*

$$\overset{\alpha}{\equiv} \exists \bar{P} \cup \hat{Q}.\mathrm{X} : [\bar{P}/P]\langle\mathcal{S}\rangle, [\bar{P}/P]\langle[\hat{Q}/Q]\langle\mathcal{S}'\rangle\rangle \tag{33}$$

(since $[\bar{P}/P]$ *bijective, and*

$\bar{P} \cap (\hat{Q} \cup \mathrm{FV}(\exists P \cup \hat{Q}.\mathrm{X} : \mathcal{S}, [\hat{Q}/Q]\langle\mathcal{S}'\rangle)) = \emptyset$

follows from (27), (24) *and* (32)*)*

$\overset{\alpha}{\equiv} \exists \bar{P} \cup \bar{Q}.\mathrm{X} : [\bar{Q}/\hat{Q}]\langle[\bar{P}/P]\langle\mathcal{S}\rangle\rangle, [\bar{Q}/\hat{Q}]\langle[\bar{P}/P]\langle[\hat{Q}/Q]\langle\mathcal{S}'\rangle\rangle\rangle$

(since $[\bar{Q}/\hat{Q}]$ *bijective, and*

$\bar{Q} \cap (\bar{P} \cup \mathrm{FV}(\exists \bar{P} \cup \hat{Q}.\mathrm{X} : [\bar{P}/P]\langle\mathcal{S}\rangle, [\bar{P}/P]\langle[\hat{Q}/Q]\langle\mathcal{S}'\rangle\rangle)) = \emptyset$

follows from (26), (25) *and* (33)*)*

$$\overset{\alpha}{\equiv} \exists \bar{P} \cup \bar{Q}.\mathrm{X} : \bar{\mathcal{S}}, \bar{\mathcal{S}}' \tag{34}$$

(by (31) *and since*

$[\bar{Q}/\hat{Q}]\langle[\bar{P}/P]\langle\mathcal{S}\rangle\rangle = [\bar{P}/P]\langle\mathcal{S}\rangle = \bar{\mathcal{S}}$

follows from (27) *and* (11) *)*

Choosing $\hat{P} \equiv \bar{P} \cup \bar{Q}$, $\hat{\mathcal{S}} \equiv \mathrm{X} : \bar{\mathcal{S}}, \bar{\mathcal{S}}'$ *and combining* (19) *with* (34) *gives the desired result.*

4.4 Conclusion

By adopting the notational changes to semantic structures and functors suggested in Section 3.3.4, and replacing the generative classification judgements by their state-less counterparts, we obtain a semantics of Mini-SML which is easily understood in terms of well-known concepts from Type Theory. Without being too precise, we will sketch the analogy between Mini-SML's semantic objects and type-theoretic constructs.

Putting aside Mini-SML's idiosyncratic denotation judgements, we find that the type theory underlying Mini-SML, embodied in its semantic objects and classification judgements, is based entirely on second-order type parameterisation and quantification, with no evidence whatsoever of first-order dependent types.

Semantic structures are related to record types: they list the types of their components. As in record types, the names of components are merely tags: they are neither free nor bound within subsequent components of the structure and there is no dependency between fields. Structures differ from record types in also recording the denotation of type components. This additional information is needed to determine the realisation of type variables when matching a structure against a signature. The enrichment relation can be seen as a combination of record subtyping and the Core subtyping relation on value types.

In Section 3.3.4, we introduced the interpretation of functors as polymorphic functions on structures returning existential structures. This interpretation is merely reinforced by the alternative presentations of the functor introduction and elimination rules (Rules (T-17) and (T-21)).

In Section 3.3.4, we also introduced the interpretation of signatures as type indexed families of structures, that is, as types parameterised by types. The alternative rules emphasise the distinct roles that signatures play in the semantics. In the functor introduction rule (Rule (T-17)), the signature is used to enforce *polymorphism*: the functor may be applied to *any* argument whose type is in the family of structures described by the signature. In the structure curtailment rule (Rule (T-22)), the signature is used to restrict the visibility and generality of the structure expression's components, by coercing its type to a *particular* member of the family of structures described by the signature. In the structure abstraction rule (Rule (T-23)), the signature is used to introduce existential quantification over types, by coercing the type of the structure expression to that of a *generic* member of the family of structures described by the signature.

Finally, in this chapter we have shown how the generative classification judgements may be regarded as a particularly operational presentation of a system based on existential quantification over types.

Although we have not gone so far as to translate Mini-SML and its semantics into an accepted type theory, we hope that the observations of the previous paragraph, together with the "type-theoretic" presentation of the judgements, give some indication of how such a translation may be achieved. Our primary motivation for the results in this chapter is to provide the necessary insight required to facilitate the extensions in subsequent chapters.

Chapter 5

Higher-Order Modules

In this chapter, we extend the Modules language of Chapter 4 to higher-order. Functors are given the same status that structures enjoy in Modules: they may be bound as components of structures, specified as functor arguments and returned as functor results. We will continue to refer to the first-order language collectively as Modules. Its generalisation will be called Higher-Order Modules. The Core language remains the same.

The chapter is organised as follows. Section 5.1 motivates the extension to higher-order with the help of an example. Section 5.2 informally explains the key ideas used to generalise the first-order static semantics we gave in Chapter 4. Section 5.3 briefly presents the phrase classes and grammar of Higher-Order Modules. In Section 5.4 we extend the definition of semantic objects to the new setting. The main difficulty is in defining a notion of enrichment between modules which is both easily understood by a programmer and a suitable basis for subtyping. We first give an intuitive, but non-definitional specification of enrichment. We then define enrichment as an inductive relation, show that is a pre-order and that it satisfies its specification. In Section 5.5 we present a static semantics for Modules. The semantics yields a type checking algorithm, provided we can give an algorithm for computing the higher-order realisations required by those rules that match semantic modules against semantic signatures. Section 5.6 presents an algorithm for computing such realisations. We prove that it is sound and complete for a restricted set of matching problems. Section 5.7 gives a brief justification of why the restricted matching algorithm may still be used to turn the static semantics into a sound and complete type checking algorithm. The work in this chapter is based, in part, on a rational reconstruction and subsequent extension of earlier research by Biswas

```
sig module Nat:sig type nat:0;
                   val z:nat;
                   val s:nat→nat;
                   val i:∀'a.'a→('a→'a)→nat→'a
               end;
    val eval:Nat.nat→(list Nat.nat)→Nat.nat
end
```

Figure 5.1: The specification of a module evaluating polynomials.

[Bis95]. Section 5.8 is a summary of our contribution and the relation to his results.

5.1 Motivation

This section presents an example program illustrating the utility of Higher-Order Modules and introducing some of the key concepts. Many more examples may be found in the literature [Tof92, Tof93, MT94, Bis95, Ler95, Ler96b, Lil97].

To help understand the example, here's a brief preview of the syntax of Higher-Order Modules. In Higher-Order Modules, the grammar of structure expressions is generalised to a syntax of *module* expressions that includes anonymous functors **functor**(X : s)m, for m a module expression, and module applications m m′. The structure definition **structure** X = s is generalised to the module definition **module** X = m, that can define a component that is either a structure or a functor. The structure specification **structure** X : S is generalised to the module specification **module** X : S that can specify a component that is either a structure or a functor. Finally, the grammar of signature expressions is extended to include *functor signatures*: informally, the functor signature **funsig**(X:S)S′ specifies the type of a functor that maps any argument matching S to some result matching S′. The meaning of these phrases will be made precise later in this chapter.

5.1.1 Programming with Higher-Order Functors

Suppose we are given the task of producing a package for evaluating polynomials over the natural numbers, where a polynomial $a_0x^0 + \cdots + a_{n-1}x^{n-1}$ is specified by the list $[a_0, \ldots, a_{n-1}]$ of its (natural) coefficients . More

```
N : sig type nat:0;
        val z:nat;
        val s:nat→nat;
        val i:∀'a.'a→('a→'a)→nat→'a
    end
```

Figure 5.2: The specification of `N`.

```
 A : funsig(X:sig type nat = N.nat;
                  val z:nat;
                  val s:nat→nat;
                  val i:∀'a.'a→('a→'a)→nat→'a
              end)
       sig val add:X.nat→X.nat→X.nat end
```

Figure 5.3: The specification of `A`.

specifically, the requirement is to produce a module matching the signature in Figure 5.1. Here `nat` is the type representing naturals, `z` is zero, `s` is the successor function on naturals, `i` implements polymorphic iteration and `eval x l` evaluates the polynomial `l` at `x`.

Let us assume we are carrying out a top-down design. We first observe that we can avoid using exponentiation in the implementation of `eval` by using Horner's rule:

$$a_0x^0 + \cdots + a_{n-1}x^{n-1} = a_0 + x(a_1 + x(\cdots + x(a_{n-1} + 0)\cdots))$$

This leaves just the implementation of the naturals, addition and multiplication to be worked out. We suspect that, given an implementation of naturals, addition should be easily defined in terms of iteration. In turn, given a means of constructing addition, multiplication should be easily obtained from iterated addition. We decide to decompose the problem into the simpler problems of implementing:

1. A structure `N` of natural numbers matching the signature in Figure 5.2.

2. A functor `A` which from `N` constructs an implementation of addition. We require that `A` matches the functor signature in Figure 5.3.

```
M : funsig(X:sig type nat = N.nat;
                 val z:nat;
                 val s:nat→nat;
                 val i:∀'a.'a→('a→'a)→nat→'a
           end)
    funsig(A:funsig(Y:sig type nat = X.nat;
                          val z:nat;
                          val s:nat→nat;
                          val i:∀'a.'a→('a→'a)→nat→'a
                      end)
                sig val add:Y.nat→Y.nat→Y.nat end)
       sig val mult:X.nat→X.nat→X.nat
       end
```

Figure 5.4: The specification of `M`.

3. A higher-order functor `M` which from `N` and `A` constructs an implementation of multiplication. We require that `M` matches the higher-order functor signature in Figure 5.4.

With higher-order functors we can delegate these subtasks to a separate team of programmers, assume implementations of `N`, `A` and `M` and implement the original specification using the higher-order functor `MkPoly` (Figure 5.5). Here, **`fix`** f takes the fix-point of a function f, implementing recursion; **`listcase`** l b f performs case analysis on the value of the list l: if this value is the empty list, b is evaluated, otherwise the function f is evaluated and applied to the head and tail of the list. Note that we can proceed with the design of `MkPoly` before modules `N`, `A` and `M` have been written.

Meanwhile, our team of programmers is busy producing prototype implementations of `N`, `A` and `M`. Fortunately, an implementation of `N` already exists as an abstract module `Nat` from which they can construct implementations of `A` and `M` (Figure 5.6).

Applying the functor `MkPoly` to `N`, `A` and `M` produces a module matching the original specification in Figure 5.1.

5.1.2 Functor Generalisation as Enrichment

During coding, the author of functor `A` realises that its body requires less structure from its argument than initially assumed. In particular, the con-

```
module MkPoly =
functor(N:sig type nat:0;
              val z:nat;
              val s:nat→nat;
              val i:∀'a.'a→('a→'a)→nat→'a
          end)
functor(A:funsig(X:sig type nat = N.nat;
                       val z:nat;
                       val s:nat→nat;
                       val i:∀'a.'a→('a→'a)→nat→'a
                   end)
          sig val add:X.nat→X.nat→X.nat end)
functor(M:funsig(X:sig type nat = N.nat;
                       val z:nat;
                       val s:nat→nat;
                       val i:∀'a.'a→('a→'a)→nat→'a
                   end)
          funsig(A:funsig(Y:sig type nat = X.nat;
                                val z:nat;
                                val s:nat→nat;
                                val i:∀'a.'a→('a→'a)→nat→'a
                             end)
                   sig val add:Y.nat→Y.nat→Y.nat end)
          sig val mult:X.nat→X.nat→X.nat  end)
  struct
    module Nat = N;
    module Add = A N;
    module Mult = M N A;
    val eval = λx.fix λevalx.λl.
          listcase l
                   (Nat.z)
                   (λh.λl. Add.add h (Mult.mult x (evalx l)))
  end
```

Figure 5.5: The implementation of MkPoly.

```
module N = Nat;

module A = functor(X:sig type nat = N.nat;
                         val z:nat;
                         val s:nat→nat;
                         val i:∀'a.'a→('a→'a)→nat→'a
                     end)
           struct val add = λn.λm. X.i n X.s m end;

module M = functor(X:sig type nat = N.nat;
                         val z:nat;
                         val s:nat→nat;
                         val i:∀'a.'a→('a→'a)→nat→'a
                     end)
           functor(A:funsig(Y:sig type nat = X.nat;
                                  val z:nat;
                                  val s:nat→nat;
                                  val i:∀'a.'a→('a→'a)→
                                            nat→'a
                              end)
                     sig val add:Y.nat→Y.nat→Y.nat end)
           struct module Add = A X;
                  val mult = λn.λm. X.i X.z (Add.add n) m
           end
```

Figure 5.6: The implementations of `N`, `A`, `M`.

```
module A' = functor(X:sig type nat:0;
                          val s:nat→nat;
                          val i: nat→(nat→nat)→nat→nat
                      end)
            struct val add = λn.λm. X.i n X.s m;
                   val sum = λb.fix λsum.λl.
                         listcase l b (λh.λl.add h (sum l))
            end
```

Figure 5.7: The functor `A'` is a more general version of `A`.

```
module N = struct type nat = int;
                  val z=0;
                  val s=λi.+ i 1;
                  val i=λb.λf.fix λibf.
                     λj.ifzero j b (f (ibf  (+ j (-1))))
           end;
module A = functor(X:sig end)
           struct val add = λi.λj.+ i j
           end;
module M = functor(X:sig end)
           functor(A:funsig(Y:sig type nat = int;
                                  val z: nat;
                                  val s: nat→nat;
                                  val i:∀'a.'a→('a→'a)→
                                            nat→'a
                              end)
                     sig end)
           struct val mult = λi.λj.* i j
           end
```

Figure 5.8: Efficient implementations of `N`, `A` and `M`.

crete representation of `X.nat` is irrelevant, no use is made of the zero component `X.z` and only a particular type instance of the iterator `X.i` is required. In the interest of writing general-purpose code (say, for inclusion in a library), he rewrites the functor as in Figure 5.7, adding, for future convenience, a function `sum` for summing over lists of naturals. From our understanding of first-order Modules, it should be clear that `A'` is more general than `A`: any argument to which `A` may be applied is also a valid argument of `A'` (but not vice-versa). Moreover, in each case, the result of applying `A'` is at least as rich as the result of applying `A` (`add` is defined and the presence of `sum` is irrelevant). Consequently, in Higher-Order Modules we will allow the application `MkPoly N A'` since the type of `A'` is at least as general as the type expected by `MkPoly N`.

5.1.3 Decomposition Need Not Compromise Efficiency

A possible objection to our choice of problem decomposition is that it sacrifices the efficiency of addition and multiplication by forcing the programmer

```
module FastNat =
 struct
    module N = ...
    module A = ...
    module M = ...
 end \
 sig
     module N: sig type nat:0;
                   val z:nat;
                   val s:nat→nat;
                   val i:∀'a.'a→('a→'a)→nat→'a
               end;
     module A:funsig(X:sig end)
                sig val add: N.nat→N.nat→N.nat end;
     module M:funsig(X:sig end)
               funsig(A:funsig(Y:sig type nat = N.nat;
                                     val z:nat;
                                     val s:nat→nat;
                                     val i:∀'a.'a→('a→'a)→
                                               nat→'a
                                 end)
                        sig end)
               sig val mult: N.nat→N.nat→N.nat end
 end
```

Figure 5.9: An efficient and abstract implementation of `N`, `A` and `M`.

to use an implementation based on iteration. This not a valid criticism, since our team can still exploit more efficient designs. For instance, assuming a built in type of integers, the team may decide to represent natural numbers as the positive subset of the integers `int` and use the built-in operations of addition `+` and multiplication `*` on integers directly, producing the code in Figure 5.8.

With the first implementation of `N`, which used the abstract Module `Nat`, we implicitly assumed that every value of type `Nat.nat` corresponded to a natural number. Since we are now using a proper subset of the integers, we should enforce the invariant that only positive integers are ever used as natural numbers. Grouping the definitions of `N`, `A` and `M` into a single module and then applying a signature abstraction to hide all occurrences of the concrete representation produces a fast implementation whose integrity cannot be violated (Figure 5.9). The application `MkPoly (FastNat.N) (FastNat.A) (FastNat.M)` is well-typed, since each of `MkPoly`'s arguments is at least as general as required. The result is an efficient implementation of polynomial evaluation.

The ability to define functors as structure components is crucial in this example. If we could only define functors at top-level, as in (first-order) Modules, we would be faced with only two design options, both of which are bad: we could either make the implementation of `N.nat` public, compromising integrity but supporting efficient implementations of `A` and `M`, or private, preserving integrity, but rendering efficient implementations of `A` and `M` impossible.

5.2 From Modules to Higher-Order Modules

5.2.1 Functor Signatures

In Higher-Order Modules, in order to define a functor taking a functor as an argument, we will need some means of indicating the formal argument's type. In (first-order) Modules, we used a signature expression to specify the structures to which a functor may be applied. Recall that a structure signature, by containing unconstrained type specifications of the form **type** t : k, will typically only specify a family of structure types. In Modules, we exploited this variation to ensure that the functor is polymorphic and may be applied to any instance of its argument signature. In Higher-Order Modules, we generalise signature expressions by introducing the functor signature phrase **funsig**(X:S)S′. By analogy to a structure signature, this phrase should specify a *family of functor types*. It is reasonable to expect each functor of a type in this family to be applicable to any module matching the argument signature S. But how do we interpret the result signature S′? Like the argument S, it may contain unconstrained type specifications of the form **type** t : k, thus specifying a family of result modules. The trick is to use this variation to define a family of functor types in the following way: the phrase **funsig**(X:S)S′ describes the family of types classifying functors that, when applied to *any* argument of a type in the family S, return a result whose type is in the family S′. The phrase denotes a family of types, indexed according to the actual realisation of type components left undetermined by **type** t : k specifications in the result signature S′.

How do we interpret this in the language of semantic objects? Before we can answer this, it helps to have an idea of what the semantic objects of Higher-Order Modules will be. We will let $\mathcal{M}, \mathcal{M}' \in \mathit{Mod}$ range over semantic modules, the disjoint union of both semantic structures $\mathcal{S} \in \mathit{Str}$ and semantic functors $\mathcal{F} \equiv \forall P.\mathcal{M}' \to \mathcal{M} \in \mathit{Fun}$. Note that, because we are in a higher-order setting, the domain and range of a semantic functor are semantic modules, not just semantic structures. Also, in Higher-Order Modules, unlike in first-order Modules, the range of a semantic functor will no longer be existentially quantified: the reason for this change will be explained in the next section. Finally, recall that, in first-order Modules, a semantic signature was just a parameterised semantic structure $\mathcal{L} \equiv \Lambda P.\mathcal{S} \in \mathit{Sig}$; in Higher-Order Modules, a semantic signature will be a parameterised semantic module $\mathcal{L} \equiv \Lambda P.\mathcal{M} \in \mathit{Sig}$, which is a natural generalisation of the first-order notion.

Now we can return to answer the question of what the semantic interpretation of the functor signature **funsig**(X:S)S′ should be. Suppose the argument signature S denotes the semantic signature $\Lambda P.\mathcal{M}$, i.e. $\mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{M}$, where P represents the type components of the argument on which any functor, of a type in this family, should behave parametrically. Since the result signature S′ may contain occurrences of X we should extend $\mathcal{C}$ by the assumption $[\mathrm{X} : \mathcal{M}]$ before elaborating S′ to its denotation (treating type variables in P as fresh parameters). Now suppose that, in this extended context, S′ denotes the semantic signature $\Lambda Q.\mathcal{M}'$, i.e. $\mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash \mathrm{S}' \triangleright \Lambda Q.\mathcal{M}'$. Here Q arises from the undetermined type components of the range and thus, by our previous discussion, should give rise to the parameters of the semantic signature of the complete phrase. Our first approximation is to say that **funsig**(X:S)S′ should denote the semantic signature $\Lambda Q.\forall P.\mathcal{M} \to \mathcal{M}'$. Unfortunately, because Q is bound before P, this approach fails to capture the functional dependency of types defined in $\mathcal{M}'$ on the parameters P. Biswas's [Bis95] important insight is to use *higher-order* type variables to encode this dependency. Let P be the set of variables $\{\alpha_0, \ldots, \alpha_{n-1}\}$. By raising the order (i.e. the kind) of each variable $\beta \in Q$, we enable it to take the parameters $\alpha_0, \ldots, \alpha_{n-1}$ as arguments. Replacing each occurrence in $\mathcal{M}'$ of $\beta \in Q$ by the application $\beta\ \alpha_0 \cdots \alpha_{n-1}$ provides for a functional dependency of these variables on the type parameters of the argument. Ignoring the usual side-conditions on bound variables, the rule relating a functor signature to its denotation can be expressed as:

$$\frac{\mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{M} \quad \mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash \mathrm{S}' \triangleright \Lambda Q.\mathcal{M}'}{\mathcal{C} \vdash \textbf{funsig}(\mathrm{X}{:}\mathrm{S})\mathrm{S}' \triangleright \Lambda \hat{Q}.\forall P.\mathcal{M} \to [\hat{Q}/Q]\,(\mathcal{M}')}$$

where $\hat{Q}$ is an appropriately raised variant of Q, and the realisation

$$[\hat{Q}/Q] = \{\beta \mapsto \beta\ \alpha_0 \cdots \alpha_{n-1} | \beta \in Q\}$$

takes care of the parameterisation (recall that $P \equiv \{\alpha_0, \ldots, \alpha_{n-1}\}$).

To use this technique, we will need to generalise our language of semantic types to the higher-order setting, obtaining a variant of the simply-typed λ-calculus. The "terms" of this calculus are semantic types extended with Λ-abstraction ($\Lambda\alpha.\tau$) and application ($\nu\ \tau$); the "types" of the calculus are the semantic kinds, extended with the higher kind ($\kappa \to \kappa'$) classifying functions on types. As in the first-order setting, realisations essentially define substitutions on type variables. However, because type variables may have higher kinds, realisations will typically be higher-order substitutions.

We illustrate the idea with a simple example. Consider the functor signature:

$$\textbf{funsig}(\textbf{X}{:}\textbf{sig type t} : \text{k}\ \textbf{end})\textbf{sig type t} : \text{k}\ \textbf{end}$$

Since

$$\ldots \vdash \textbf{sig type t} : \text{k}\ \textbf{end} \triangleright \Lambda\{\alpha\}.(\textbf{t} = \alpha),$$

and

$$\ldots[\textbf{X} : (\textbf{t} = \alpha)] \vdash \textbf{sig type t} : \text{k}\ \textbf{end} \triangleright \Lambda\{\beta\}.(\textbf{t} = \beta),$$

from our previous discussion it should be clear that, by raising β to account for dependencies on α:

$$\begin{aligned}\ldots \vdash \textbf{funsig}(\textbf{X}{:}\textbf{sig type t} : \text{k}\ \textbf{end})\textbf{sig type t} : \text{k}\ \textbf{end} \triangleright \\ \Lambda\{\beta\}.\forall\{\alpha\}.(\textbf{t} = \alpha) \to (\textbf{t} = \beta\ \alpha).\end{aligned} \qquad (*)$$

Let's consider some examples of functors that match signature (*). The identity functor

$$\textbf{functor}(\textbf{X} : \textbf{sig type t} : \text{k}\ \textbf{end})\textbf{X}$$

has module type

$$\forall\{\alpha\}.(\textbf{t} = \alpha) \to (\textbf{t} = \alpha),$$

which matches the signature (*) by choosing the higher-order realisation $[\Lambda\alpha.\alpha/\beta]$.

The constant functor

$$\textbf{functor}(\textbf{X} : \textbf{sig type t} : \text{k}\ \textbf{end})\textbf{struct type t} = \textbf{int end},$$

has module type

$$\forall\{\alpha\}.(\mathbf{t} = \alpha) \to (\mathbf{t} = \mathbf{int}),$$

which matches the signature (*) by choosing the realisation $[\Lambda\alpha.\mathbf{int}/\beta]$.

The functor

$$\begin{array}{l} \textbf{module F} = \\ \quad \textbf{functor}(\mathbf{X} : \textbf{sig type t} : \mathrm{k}\ \textbf{end})\textbf{struct type t} = \mathbf{X.t} \to \mathbf{X.t}\ \textbf{end} \end{array}$$

has module type

$$\forall\{\alpha\}.(\mathbf{t} = \alpha) \to (\mathbf{t} = \alpha \to \alpha),$$

which matches the signature (*) by choosing the realisation $[\Lambda\alpha.\alpha \to \alpha/\beta]$.

Now consider the higher-order functor:

$$\begin{array}{l} \textbf{module H} = \\ \qquad \textbf{functor}(\mathbf{G} : \textbf{funsig}(\mathbf{X}\text{:}\textbf{sig type t} : \mathrm{k}\ \textbf{end})\textbf{sig type t} : \mathrm{k}\ \textbf{end}) \\ \qquad\qquad \textbf{functor}(\mathbf{Y} : \textbf{sig type t} : \mathrm{k}\ \textbf{end}) \\ \qquad\qquad\qquad \mathbf{G}\ (\mathbf{G}\ \mathbf{Y}). \end{array}$$

It has module type:

$$\begin{array}{l} \forall\{\beta\}.(\forall\{\alpha\}.(\mathbf{t} = \alpha) \to (\mathbf{t} = \beta\ \alpha)) \to \\ \qquad\quad \forall\{\delta\}.(\mathbf{t} = \delta) \to \\ \qquad\qquad\qquad (\mathbf{t} = \beta\ (\beta\ \delta)). \end{array}$$

Notice how **H** is polymorphic in its first argument's (i.e. **G**'s) argument-result dependency β, and that this argument is itself required to be polymorphic in α. Moreover, the argument's polymorphism is actually exploited within the functor body, since **G** is applied at two different instances, choosing $[\delta/\alpha]$ and $[\beta\ \delta/\alpha]$ respectively. Since **F** matches **H**'s argument signature (via the realisation $[\Lambda\alpha.\alpha \to \alpha/\beta]$), the partial application **H F** has module type:

$$\forall\{\delta\}.(\mathbf{t} = \delta) \to (\mathbf{t} = (\delta \to \delta) \to (\delta \to \delta)).$$

5.2.2 Incorporating Generativity

In the previous discussion, no mention was made of the generative nature of functors. Recall that in first-order Modules, a functor may return new types as the result of abstractions and functor applications appearing within its body; in order to preserve type soundness, each application of a given functor causes the generation of fresh types. This "generative" capability is reflected in the semantic objects classifying functors: a semantic functor has the form $\forall P.\mathcal{S} \to \exists Q.\mathcal{S}'$, where generativity is captured by the existential quantification of Q in the result. In the examples of the previous section we implicitly assumed that semantic functors had the form $\forall P.\mathcal{M} \to \mathcal{M}'$, where the result $\mathcal{M}'$ is a simple module (either a structure or functor), and not, conspicuously, an existential module. Indeed, in his paper, Biswas explicitly uses the simplifying assumption that generativity has been removed from the language [Bis95].

While it is possible to do this, the practical ramifications for programming in the language are rather severe. To ensure data abstraction (one of the key motivations for using a modules language), programs have to be written in a fully functorised form. By this we mean the following. Suppose $P[\mathrm{m}]$ is a program with an occurrence of a module m, implementing the signature S, and we wish to ensure that the module expression m can be replaced by any other implementation m' matching S. Using an abstraction, we can isolate m from its context P by writing $P[\mathrm{m} \setminus \mathrm{S}]$. If this program type-checks, then so does $P[\mathrm{m}' \setminus \mathrm{S}]$. If we remove generativity from the language then we can no longer accommodate the abstraction phrase. Without abstractions, we can only ensure the above property if P is written as the *outermost* application of a functor to m, i.e. if P has the form $(\mathbf{functor}(\mathbf{X} : \mathrm{S})\mathrm{m}'')$ $[\mathrm{m}]$. Notice that if the application is not outermost, i.e. P is merely in the form $P'[(\mathbf{functor}(\mathbf{X} : \mathrm{S})\mathrm{m}'')$ $[\mathrm{m}]]$ for a non-empty program context P', then the inner functor application may propagate the actual implementations of types in m that are meant to be abstract according to the signature S. With access to the concrete implementations, the outer context P' can inadvertently make use of this information and violate the intended abstraction, preventing the replacement of m by m'. Unfortunately, insisting on fully functorised code leads to an unnatural and unintelligible coding style in which all abstract modules must be anticipated early on and imported as initial functor arguments, possibly at considerable distance from their point of use. As MacQueen [Mac86] rightly points out, this seriously impedes the incremental construction of programs, which is, after all, the main motivation for using a modules language. Notice, also, that the approach only works if both S and m are closed[1], since each must be well-formed in the outermost and thus empty context. An abstraction phrase, on the other hand, may be embedded deep within a program, and may be used to isolate an open[2] module expression using an open signature.

Biswas leaves the extension of his proposal to handle generativity as an "important direction for future research associated with this approach to providing semantics to higher-order functors" [Bis95]. He does not give any concrete indication of how this may be accomplished beyond speculating that "by considering *gensym* as a primitive function and introducing environments, can we capture some form of generativity in the language" [Bis95]. The fact that this statement was written nearly five years after the publication of the Definition of Standard ML [MTH90] is evidence of the prevalent, state-based understanding of generativity. With our understanding of generativity as existential quantification, we shall see that the higher-order extension is almost trivial. This application alone hopefully justifies the pedantic but nevertheless useful reformulation of the static semantics which we undertook in Chapter 4.

Given that we want some notion of generativity in Higher-Order Modules, we actually have two ways to proceed. We could attempt to extend Biswas's approach, discussed in the previous section, by adding existential quantification to his semantic functors. This would be a considerable departure from his work and it is not clear whether it would succeed. At the very least, the syntax of functor result signatures has to be extended to allow the specification of generative as well as undetermined types. Furthermore, the notion of enrichment between functors has to be altered in a non-trivial manner to take account of existentially quantified results.

Fortunately, there is a much simpler approach: we can relax the notion of functor generativity in a way that eliminates the need for existentially quantifying over a functor's result type. We encountered the germ of this idea in Chapter 4, Section 4.1.3, where we briefly considered the consequences of making functors *applicative*. The suggestion was to do away with the generation of fresh types at each application of a given functor, by, instead, generating fresh types once and for all at the functor's point of definition. The term "applicative" refers to the property that two distinct applications of the same functor will yield equivalent abstract types. Expressed in terms of existential quantification and ignoring the usual side-conditions preventing variable capture, the proposal meant replacing the "generative" functor

[1]A phrase is *closed* if it contains no free identifiers.

[2]A phrase is *open* if it contains zero or more free identifiers.

introduction and elimination rules:

$$\frac{\mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{S} \quad \mathcal{C}[\mathrm{X} : \mathcal{S}] \vdash \mathrm{s} : \exists Q.\mathcal{S}' \quad \mathcal{C}[\mathrm{F} : \forall P.\mathcal{S} \to \exists Q.\mathcal{S}'] \vdash \mathrm{b} : \exists Q'.\mathcal{S}''}{\mathcal{C} \vdash \textbf{functor}\ \mathrm{F}\ (\mathrm{X} : \mathrm{S})\ =\ \mathrm{s}\ \textbf{in}\ \mathrm{b} : \exists Q'.\mathcal{S}''} \tag{T-17}$$

$$\frac{\begin{array}{ll}\mathcal{C}(\mathrm{F}) = \forall P.\mathcal{S}' \to \exists Q.\mathcal{S} & \mathcal{C} \vdash \mathrm{s} : \exists Q'.\mathcal{S}'' \\ \mathcal{S}'' \succeq \varphi(\mathcal{S}') & \mathrm{Dom}(\varphi) = P\end{array}}{\mathcal{C} \vdash \mathrm{F}\ \mathrm{s} : \exists Q \cup Q'.\varphi(\mathcal{S})} \tag{T-21}$$

by the “applicative” rules:

$$\frac{\mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{S} \quad \mathcal{C}[\mathrm{X} : \mathcal{S}] \vdash \mathrm{s} : \exists Q.\mathcal{S}' \quad \mathcal{C}[\mathrm{F} : \forall P.\mathcal{S} \to \mathcal{S}'] \vdash \mathrm{b} : \exists Q'.\mathcal{S}''}{\mathcal{C} \vdash \textbf{functor}\ \mathrm{F}\ (\mathrm{X} : \mathrm{S})\ =\ \mathrm{s}\ \textbf{in}\ \mathrm{b} : \exists Q \cup Q'.\mathcal{S}''} \tag{*}$$

$$\frac{\begin{array}{ll}\mathcal{C}(\mathrm{F}) = \forall P.\mathcal{S}' \to \mathcal{S} & \mathcal{C} \vdash \mathrm{s} : \exists Q'.\mathcal{S}'' \\ \mathcal{S}'' \succeq \varphi(\mathcal{S}') & \mathrm{Dom}(\varphi) = P\end{array}}{\mathcal{C} \vdash \mathrm{F}\ \mathrm{s} : \exists Q'.\varphi(\mathcal{S})} \tag{T-21'}$$

Observe that the applicative functor introduction rule (Rule (*)) eliminates the existential quantification of Q at F's point of definition *before* proceeding with the classification of b; Q is added once and for all to the set of existential types produced by the classification of the complete phrase. As a result, the elimination rule (Rule (T-21')) is simpler: only the argument of an application, not its functor, will introduce existential variables. Clearly, with the applicative rules, the general form of semantic functors can now be simplified to $\forall P.\mathcal{S} \to \mathcal{S}'$, dropping any existential quantification over the result. This then allows us to apply Biswas's results directly. Unfortunately, as demonstrated by the counter-example in Chapter 4, Figure 4.4(c), Rule (*) is not sound. The types hidden by Q may, in general, have a functional dependency on the type parameters P of the functor. Directly eliminating the existential from the range signature $\mathcal{S}'$ ignores this dependency.

All is not lost however. Resorting to higher-order type variables, we can encode such functional dependencies by exploiting essentially the idea used to define the interpretation of functor signatures. Let P be the set of variables $\{\alpha_0, \ldots, \alpha_{n-1}\}$. By raising the order of each variable $\beta \in Q$, we enable it to take the parameters $\alpha_0, \ldots, \alpha_{n-1}$ as arguments. Replacing each occurrence in the functor range $\mathcal{S}'$ of $\beta \in Q$ by the application $\beta\ \alpha_0 \cdots \alpha_{n-1}$ provides for the functional dependency of these variables on the type parameters of the functor.

We can now formulate a sound introduction rule:

$$\frac{\begin{array}{c}\mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{S} \quad \mathcal{C}[\mathrm{X} : \mathcal{S}] \vdash \mathrm{s} : \exists Q.\mathcal{S}' \\ \mathcal{C}[\mathrm{F} : \forall P.\mathcal{S} \rightarrow [\hat{Q}/Q]\,(\mathcal{S}')] \vdash \mathrm{b} : \exists Q'.\mathcal{S}''\end{array}}{\mathcal{C} \vdash \textbf{functor}\ \mathrm{F}\ (\mathrm{X} : \mathrm{S})\ =\ \mathrm{s}\ \textbf{in}\ \mathrm{b} : \exists \hat{Q} \cup Q'.\mathcal{S}''} \quad (\text{T-17'})$$

where $\hat{Q}$ is an appropriately raised variant of Q and the realisation

$$[\hat{Q}/Q] = \{\beta \mapsto \beta\ \alpha_0 \cdots \alpha_{n-1} | \beta \in Q\}$$

takes care of the parameterisation, provided $P \equiv \{\alpha_0, \ldots, \alpha_{n-1}\}$. In effect, this amounts to the *skolemisation* of the existentially quantified variables Q by the universally quantified variables P.

In the sound applicative semantics, the term “applicative” refers to the property that two distinct applications of the same functor will yield equivalent abstract types, *provided they both agree on the realisation of the functor's type parameters.*

The example in Figure 5.10, continued in Figure 5.11, illustrates the differences between adopting a generative semantics, naive applicative semantics and sound applicative semantics for functors.

5.2.3 Generalising the Dot Notation

In Modules, the dot notation, used to project components from structures, is syntactically restricted to *paths* sp $\in$ StrPath. Recall that a path is essentially a non-empty, dot-separated sequence of structure identifiers. As a result, we can only access components of named, but not anonymous, structure expressions. As observed by Leroy [Ler95], if we retain this syntactic restriction in Higher-Order Modules, it becomes impossible to fully specify the types returned by functor applications. Consider the example of wanting to express a functor which, given two functors `H` and `G` returning types `u` and `v` respectively, returns a functor that constructs a derived operation that requires the compatibility of these types:

$$
\begin{array}{l}
\mathbf{functor}\ \mathbf{F}(\mathbf{X}\colon\ \mathbf{sig\ type\ t} : 0\ \mathbf{end}) = \\
\qquad\qquad \mathbf{struct\ datatype\ u}\ =\ \mathbf{X}.\mathbf{t}\ \mathbf{with}\ \mathbf{x}, \mathbf{y}\ \mathbf{end} \\
\mathbf{in} \\
\mathbf{structure}\ \mathbf{X}\ =\ \mathbf{F}\ (\mathbf{struct\ type\ t} = \mathbf{int\ end}); \\
\mathbf{structure}\ \mathbf{Y}\ =\ \mathbf{F}\ (\mathbf{struct\ type\ t} = \mathbf{int\ end}); \\
\mathbf{structure}\ \mathbf{Z}\ =\ \mathbf{F}\ (\mathbf{struct\ type\ t} = \mathbf{int} \to \mathbf{int\ end}); \\
\mathbf{val\ x} = \mathbf{X}.\mathbf{y}\ (\mathbf{Y}.\mathbf{x}\ 1); \\
\mathbf{val\ z} = (\mathbf{Z}.\mathbf{y}\ (\mathbf{Y}.\mathbf{x}\ 1))\ 2
\end{array}
$$

(a) The definition of **x** is sound. The definition of **z** is not, attempting to apply 1 to 2. It should be rejected by a sound static semantics.

$$
\begin{array}{l}
\mathbf{functor}\ \lfloor_{\forall\{\alpha\}.(\mathbf{t}=\alpha)\to\exists\{\beta\}.(\mathbf{u}=\beta,\mathbf{x}:\alpha\to\beta,\mathbf{y}:\beta\to\alpha)}\mathbf{F}\ (\mathbf{X}\colon\ \mathbf{sig\ type\ t} : 0\ \mathbf{end}) = \\
\qquad\qquad \mathbf{struct\ datatype\ u}\ =\ \mathbf{X}.\mathbf{t}\ \mathbf{with}\ \mathbf{x}, \mathbf{y}\ \mathbf{end} \\
\mathbf{in} \\
\mathbf{structure}\ \lfloor_{(\mathbf{u}=\beta,\mathbf{x}:\mathbf{int}\to\beta,\mathbf{y}:\beta\to\mathbf{int})}\mathbf{X}\ =\ \mathbf{F}\ (\mathbf{struct\ type\ t} = \mathbf{int\ end}); \\
\mathbf{structure}\ \lfloor_{(\mathbf{u}=\delta,\mathbf{x}:\mathbf{int}\to\delta,\mathbf{y}:\delta\to\mathbf{int})}\mathbf{Y}\ =\ \mathbf{F}\ (\mathbf{struct\ type\ t} = \mathbf{int\ end}); \\
\mathbf{structure}\ \lfloor_{(\mathbf{u}=\gamma,\mathbf{x}:(\mathbf{int}\to\mathbf{int})\to\gamma,\mathbf{y}:\gamma\to(\mathbf{int}\to\mathbf{int}))}\mathbf{Z}\ =\ \mathbf{F}\ (\mathbf{struct\ type\ t} = \mathbf{int} \to \mathbf{int\ end}); \\
\mathbf{val\ x} = \underline{\mathbf{X}.\mathbf{y}_{\beta\to\mathbf{int}}\ (\mathbf{Y}.\mathbf{x}_{\mathbf{int}\to\delta}\ 1)_{\delta}}; \\
\mathbf{val\ z} = \underline{(\mathbf{Z}.\mathbf{y}_{\gamma\to(\mathbf{int}\to\mathbf{int})}\ (\mathbf{Y}.\mathbf{x}_{\mathbf{int}\to\delta}\ 1)_{\delta})}\ 2
\end{array}
$$

(b) The partial, unsuccessful classification of the phrase in Figure 5.10(a) using the standard, generative semantics (Rules (T-17) and (T-21)). Notice how the generation of fresh types at each and every functor application ensures that **X**.**u**, **Y**.**u** and **Z**.**u** are all distinct, preserving soundness. The offending subphrases are underlined.

Figure 5.10: A phrase illustrating the difference between generative and applicative functors.

$$\mathbf{functor}\ \lfloor\mathbf{F}_{\forall\{\alpha\}.(\mathbf{t}=\alpha)\to(\mathbf{u}=\beta,\mathbf{x}:\alpha\to\beta,\mathbf{y}:\beta\to\alpha)}\ (\mathbf{X}\!:\ \mathbf{sig\ type\ t} : 0\ \mathbf{end}) =$$
$$\mathbf{struct\ datatype\ u}\ =\ \mathbf{X}.\mathbf{t}\ \mathbf{with}\ \mathbf{x}, \mathbf{y}\ \mathbf{end}$$
$$\mathbf{in}$$
$$\mathbf{structure}\ \lfloor\mathbf{X}_{(\mathbf{u}=\beta,\mathbf{x}:\mathbf{int}\to\beta,\mathbf{y}:\beta\to\mathbf{int})}\ =\ \mathbf{F}\ (\mathbf{struct\ type\ t} = \mathbf{int\ end});$$
$$\mathbf{structure}\ \lfloor\mathbf{Y}_{(\mathbf{u}=\beta,\mathbf{x}:\mathbf{int}\to\beta,\mathbf{y}:\beta\to\mathbf{int})}\ =\ \mathbf{F}\ (\mathbf{struct\ type\ t} = \mathbf{int\ end});$$
$$\mathbf{structure}\ \lfloor\mathbf{Z}_{(\mathbf{u}=\beta,\mathbf{x}:(\mathbf{int}\to\mathbf{int})\to\beta,\mathbf{y}:\beta\to(\mathbf{int}\to\mathbf{int}))}\ =\ \mathbf{F}\ (\mathbf{struct\ type\ t} = \mathbf{int} \to \mathbf{int\ end});$$
$$\mathbf{val\ x} = (\mathbf{X}.\mathbf{y}_{\beta\to\mathbf{int}}\ (\mathbf{Y}.\mathbf{x}_{\mathbf{int}\to\beta}\ 1)_{\beta})_{\mathbf{int}};$$
$$\mathbf{val\ z} = (\mathbf{Z}.\mathbf{y}_{\beta\to(\mathbf{int}\to\mathbf{int})}\ (\mathbf{Y}.\mathbf{x}_{\mathbf{int}\to\beta}\ 1)_{\beta})_{\mathbf{int}\to\mathbf{int}}\ 2$$

(a) A completely successful but unsound classification of the phrase in Figure 5.10(a), constructed in a semantics employing naive applicative functors (Rules (*) and (T-21')). **X**.**u**, **Y**.**u** and **Z**.**u** are incorrectly identified.

$$\mathbf{functor}\ \lfloor\mathbf{F}_{\forall\{\alpha\}.(\mathbf{t}=\alpha)\to(\mathbf{u}=\beta\ \alpha,\mathbf{x}:\alpha\to\beta\ \alpha,\mathbf{y}:\beta\ \alpha\to\alpha)}\ (\mathbf{X}\!:\ \mathbf{sig\ type\ t} : 0\ \mathbf{end}) =$$
$$\mathbf{struct\ datatype\ u}\ =\ \mathbf{X}.\mathbf{t}\ \mathbf{with}\ \mathbf{x}, \mathbf{y}\ \mathbf{end}$$
$$\mathbf{in}$$
$$\mathbf{structure}\ \lfloor\mathbf{X}_{(\mathbf{u}=(\beta\ \mathbf{int}),\mathbf{x}:\mathbf{int}\to(\beta\ \mathbf{int}),\mathbf{y}:(\beta\ \mathbf{int})\to\mathbf{int})}\ =\ \mathbf{F}\ (\mathbf{struct\ type\ t} = \mathbf{int\ end});$$
$$\mathbf{structure}\ \lfloor\mathbf{Y}_{(\mathbf{u}=(\beta\ \mathbf{int}),\mathbf{x}:\mathbf{int}\to(\beta\ \mathbf{int}),\mathbf{y}:(\beta\ \mathbf{int})\to\mathbf{int})}\ =\ \mathbf{F}\ (\mathbf{struct\ type\ t} = \mathbf{int\ end});$$
$$\mathbf{structure}\ \lfloor\mathbf{Z}_{(\mathbf{u}=(\beta\ (\mathbf{int}\to\mathbf{int})),\mathbf{x}:(\mathbf{int}\to\mathbf{int})\to(\beta\ (\mathbf{int}\to\mathbf{int})),\mathbf{y}:(\beta\ (\mathbf{int}\to\mathbf{int}))\to(\mathbf{int}\to\mathbf{int}))}\ =\ \mathbf{F}\ (\mathbf{struct\ type\ t} = \mathbf{int} \to \mathbf{int\ end});$$
$$\mathbf{val\ x} = (\mathbf{X}.\mathbf{y}_{(\beta\ \mathbf{int})\to\mathbf{int}}\ (\mathbf{Y}.\mathbf{x}_{\mathbf{int}\to(\beta\ \mathbf{int})}\ 1)_{(\beta\ \mathbf{int})})_{\mathbf{int}};$$
$$\mathbf{val\ z} = \underline{(\mathbf{Z}.\mathbf{y}_{(\beta\ (\mathbf{int}\to\mathbf{int}))\to(\mathbf{int}\to\mathbf{int})}\ (\mathbf{Y}.\mathbf{x}_{\mathbf{int}\to(\beta\ \mathbf{int})}\ 1)_{(\beta\ \mathbf{int})})}\ 2$$

(b) An incomplete but sound classification of the phrase in Figure 5.10(a), constructed in a semantics using the correct rules for applicative functors (Rules (T-17') and (T-21')). Even though **X**.**u** and **Y**.**u** are (safely) identified, they are still correctly distinguished from **Z**.**u**.

Figure 5.11: Classifying the phrase in Figure 5.10(a) in both a naive applicative semantics and a sound applicative semantics.

```
functor(H:funsig(X:sig type t:0 end)
           sig type u : 0;
               val in : X.t → u
           end)
functor(G:funsig(X:sig type t:0 end)
            sig type v = [?];
                val out : v → X.t
            end)
functor(X:sig type t:0 end)
   struct  module Y = H X;
           module Z = G X;
           val image = λx. Z.out (Y.in x)
   end
```

Of course, this program fails to type check unless we can specify that the types returned by `H` and `G` are compatible by filling in the $\boxed{?}$ in the definitional specification of `v`. Unfortunately, if we restrict projections to paths, our only option is to fix not only `v` but also `u`, by giving equivalent concrete definitions with specifications of the form `type u =` d and `type v =` d′ for some particular definitions d and d′ denoting the same type *d*. If `u` is not fixed, there is no syntax to express that for every argument `X`, the `v` component of `(G X)` should be equivalent to the `u` component of `(H X)`. Intuitively, however, the functor body should type-check for *any* definition of `u`, provided `u` and `v` are equivalent as functions of `X.t`. If we generalise the dot notation to operate on *arbitrary* module expressions m, allowing module, type and value projections of the form m.X, m.t and m.x, then we can replace the $\boxed{?}$ by the type projection `(H X).u`, yielding a functor that is polymorphic in the definition of `u`, capturing our intuition.

Remark 5.2.1 *(For Type Theorists).* There is also a more theoretical motivation for generalising projections. If we want to prove a syntactic *subject reduction* result for Modules, then a key lemma we will need is that the type of a functor application is preserved when substituting the actual argument for the formal argument of the functor. It is easy to see that the phrase class StrPath is not even syntactically closed under substitution of module phrases for identifiers, making it impossible to state this lemma, let alone prove it. By generalising projections from paths to projections from arbitrary module expressions, the syntax of Modules becomes closed under substitution, bringing us one step closer[3] to proving syntactic subject reduc-

[3]But not quite all the way there, for reasons we shall not explore further in this thesis.

t ∈ TypId	type identifiers
x ∈ ValId	value identifiers
X ∈ ModId	*module identifiers*

(a) Identifiers

B ∈ SigBod	signature bodies
S ∈ SigExp	signature expressions
do ∈ TypOcc	type occurrences

(b) Type Syntax

b ∈ StrBod	structure bodies
m ∈ ModExp	*module expressions*
vo ∈ ValOcc	value occurrences

(c) Term Syntax

Figure 5.12: Higher-Order Modules Phrase Classes

tion. Courant [Cou97b] addresses a similar failing of Leroy's module calculi: in Leroy's original proposal [Ler94], projections are restricted to paths; even Leroy's extended notion of path [Ler95], that includes applications of (functor) paths to (argument) paths, fails to remedy this problem with subject reduction.

5.3 Phrase Classes

Figure 5.12 presents the phrase classes of Higher-Order Modules. Figure 5.13 defines their (abstract) grammar. Most of the phrases in Higher-Order Modules should be familiar from their counterparts in first-order Modules. We will focus our attention on the differences between the grammars.

Identifiers X ∈ ModId range over module expressions and subsume the separate phrase classes StrId and FunId of Modules. The phrase class ModExp generalises the first-order phrase class StrExp of structure expressions. Phrases m ∈ ModExp are used to express both structures and functors. The distinguished phrase class of structure paths sp ∈ StrPath for accessing subcomponents of structures has been removed. Instead, we extend the class of module expressions, type occurrences and value occurrences

TypId	$\overset{\text{def}}{=}$	$\{\mathbf{t}, \mathbf{u}, \ldots\}$	type identifiers
ValId	$\overset{\text{def}}{=}$	$\{\mathbf{x}, \mathbf{y}, \ldots\}$	value identifiers
ModId	$\overset{\text{def}}{=}$	$\{\mathbf{X}, \mathbf{Y}, \mathbf{F}, \mathbf{G}, \ldots\}$	*module identifiers*
B	::=	**type** t = d; B	type definition
	|	**type** t : k; B	type specification
	|	**val** x : v; B	value specification
	|	**module** X : S; B	*module specification*
	|	ϵ_{B}	empty body
S	::=	**sig** B **end**	structure signature
	|	**funsig**(X:S)S′	*functor signature*
do	::=	t	type identifier
	|	m.t	*type projection*
b	::=	**type** t = d; b	type definition
	|	**val** x = e; b	value definition
	|	**module** X = m; b	*module definition*
	|	**local** X = m **in** b	*local module definition*
	|	ϵ_{b}	empty body
m	::=	X	*module identifier*
	|	m.X	*submodule projection*
	|	**struct** b **end**	structure
	|	**functor**(X : S)m	*functor*
	|	m m′	*functor application*
	|	$\mathrm{m} \succeq \mathrm{S}$	signature curtailment
	|	$\mathrm{m} \setminus \mathrm{S}$	signature abstraction
vo	::=	x	value identifier
	|	m.x	*value projection*

Figure 5.13: Higher-Order Modules Grammar

with generalised projections. The class StrPath is redundant since we essentially have StrPath $\subseteq$ ModExp.

Signature bodies B $\in$ SigBod are defined as for first-order Modules, except that we replace the structure specification **structure** X : S; B by its generalisation **module** X : S; B. The phrase specifies a module named X, matching the signature S. Note that S may specify either a structure or a functor.

Signature expressions S $\in$ SigExp specify modules. The new phrase **funsig**(X:S)S$'$ specifies a functor with argument signature S and result signature S$'$. X is bound in S$'$. In particular, types defined in S$'$ may refer to X. Moreover, types merely specified, but not defined, in S$'$ have an implicit dependency on X.

Structure bodies b $\in$ StrBod are defined as in (first-order) Modules, except that we replace the structure definitions **structure** X = s;b and **local** X = s **in** b by their generalisations **module** X = m; b and **local** X = m **in** b. The phrase **module** X = m; b defines X as a component of the surrounding structure expression that can be accessed by the dot-notation. Since m may be a functor, structures may now contain functor components. Recall that Modules only catered for local definitions of functors; this restriction has now been removed. The corresponding phrase **functor** F (X : S) = m **in** b is redundant and has been deleted, since it can be treated as an abbreviation of **local** F = **functor**(X : S)m **in** b.

Module expressions m $\in$ ModExp evaluate to both structures, i.e. collections of type, value and module definitions, and functors. Every module identifier X is a proper module expression, as is the direct projection m.X of the submodule X from m (provided m evaluates to a structure). The phrase **struct** b **end** encapsulates a structure body to form a module. The phrase **functor**(X : S)m is an anonymous functor, i.e. a parameterised module. The identifier X names the formal argument. The scope of X is the functor body m. The functor may be applied to any module that matches the argument's signature S. Note that S may specify either a functor or a structure, and that m may itself evaluate to either a functor or a structure. The phrase m m$'$ is the application of the (possibly anonymous) module m, which must be a functor, to the module m$'$. Since functors may now take functors as arguments, m$'$ may itself be a functor. Curtailments and abstractions have the same interpretation as in the first-order setting. The phrase m $\succeq$ S matches the module m against the signature S and curtails it accordingly: m is specialised according to S, provided the type of m enriches a suitable realisation of S. The actual realisation of types that are specified, but not defined, in S is retained. Note that if m is a functor and S a functor signature, then the curtailment preserves the actual argument-result dependencies of m that are merely specified, but not defined, in the functor signature S. The abstraction m $\setminus$ S is similar to the curtailment m $\succeq$ S. However, the actual realisation of types is *hidden* outside the abstraction. Note that if m is a functor and S a functor signature, then the abstraction hides the actual argument-result dependencies of m. (In the static semantics, abstractions will just introduce existentially quantified semantic modules, just as they introduced existentially quantified semantic structures in the first-order semantics of Chapter 4.)

Informally, a module expression *matches* a signature as follows. If m is a module evaluating to a structure, then m matches S provided that S is of the form **sig** B **end**, and, as in the first-order setting, it implements all of the components specified in the signature body B. In particular, the structure must *realise* all of the type components that are merely specified but not defined in B. Moreover, the structure must *enrich* B subject to this realisation: every specified type must be implemented by an equivalent type; every specified value must be implemented by a value whose type is at least as general as its specification; finally, every specified module must be implemented by a module that enriches its specification. As before, the order in which components of the structure are actually defined is irrelevant. Furthermore, the structure is free to define more components than are specified in the signature. If m is a module evaluating to a functor, then m matches S provided S is of the form **funsig**(X:S$'$)S$''$, and there is a realisation of the argument-result dependencies of S(i.e. the dependencies of types merely specified in S$''$ on types merely specified in S$'$), such that, whenever an actual argument m$'$ matches the signature S$'$, then the application m m$'$ evaluates to a module matching the realisation of S$''$. Of course, we won't need to evaluate module expressions to check matching; it will be enough to know the type of the module at hand. This will be all be made more precise in Section 5.4.

Finally, the grammars of type occurrences do $\in$ TypOcc and value occurrences vo $\in$ ValOcc are modified, replacing restricted type and value projections sp.t and sp.x by the generalised phrases m.t and m.x respectively. Of course, the static semantics will have to ensure that the module expression m evaluates to a structure and not a functor.

$\kappa \in \mathit{Kind}$	kinds classifying types
$\alpha^\kappa \in \mathit{TypVar}^\kappa$	type variables
$M,N,P,Q,R \in \mathit{TypVarSet}$	variable sets
$\nu^\kappa \in \mathit{TypNam}^\kappa$	type names
$\tau^\kappa \in \mathit{Typ}^\kappa$	types
$\mathcal{S} \in \mathit{Str}$	structures
$\mathcal{F} \in \mathit{Fun}$	functors
$\mathcal{M} \in \mathit{Mod}$	*modules*
$\mathcal{X} \in \mathit{ExMod}$	existential *modules*
$\mathcal{L} \in \mathit{Sig}$	signatures
$\mathcal{C} \in \mathit{Context}$	contexts

Figure 5.14: Semantic Objects of Higher-Order Modules

5.4 Semantic Objects

Figures 5.14, 5.15 and 5.16 define the semantic objects assigned to module expressions. They serve the role of types in the module semantics. We let $\mathcal{O}$ range over all semantic objects.

Definition 5.1 (Kinds, Type Variables, Type Names and Types).
A kind $\kappa \in \mathit{Kind}$ is either a Core kind $\mathrm{k} \in \mathrm{DefKind}$ used to specify definable types, or a *higher* kind $\kappa \to \kappa'$, classifying *functions* from types of kind κ to types of kind κ'.

Kinds are used to index sets of kind-equivalent type variables, type names and types. For each kind $\kappa \in \mathit{Kind}$ we have:

- An infinite, denumerable set of *type variables*, TypVar^κ. A type variable $\alpha^\kappa \in \mathit{TypVar}^\kappa$ ranges over types in Typ^κ.

- A set of *type names*, TypNam^κ. A *type name* $\nu^\kappa \in \mathit{TypNam}^\kappa$ is either a type variable of kind κ, or an *application* $\nu'\ \tau$ of a type name ν' of kind $\kappa' \to \kappa$ to a type τ of kind κ'.

$\kappa \in \mathit{Kind}$	$::=$	k	Core kind
	$\mid$	$\kappa \rightarrow \kappa'$	*function space*
$\alpha^{\kappa} \in \mathit{TypVar}^{\kappa}$	$\stackrel{\mathrm{def}}{=}$	$\{\alpha^{\kappa}, \beta^{\kappa}, \delta^{\kappa}, \gamma^{\kappa}, \ldots\}$	an infinite, denumerable set
$\alpha \in \mathit{TypVar}$	$\stackrel{\mathrm{def}}{=}$	$\biguplus_{\kappa \in \mathit{Kind}} \mathit{TypVar}^{\kappa}$	
$P \in \mathit{TypVarSet}$	$\stackrel{\mathrm{def}}{=}$	$\mathrm{Fin}(\mathit{TypVar})$	
$\nu^{\kappa} \in \mathit{TypNam}^{\kappa}$	$::=$	α^{κ}	type variable
	$\mid$	$\nu^{\kappa' \rightarrow \kappa}\ \tau^{\kappa'}$	*type application*
$\tau^{\kappa} \in \mathit{Typ}^{\kappa}$	$::=$	d^{k} (provided $\kappa \equiv \mathrm{k} \in \mathrm{DefKind}$)	Core definable type
	$\mid$	$\Lambda\alpha^{\kappa'}.\tau^{\kappa''}$ (provided $\kappa \equiv \kappa' \rightarrow \kappa''$)	*type function*
	$\mid$	ν^{κ}	type name
$\tau \in \mathit{Typ}$	$\stackrel{\mathrm{def}}{=}$	$\biguplus_{\kappa \in \mathit{Kind}} \mathit{Typ}^{\kappa}$	

Figure 5.15: Semantic Objects of Higher-Order Modules (cont.)

$\mathcal{S} \in \mathit{Str}$	$::=$	$\mathrm{t} = \tau^{\mathrm{k}}, \mathcal{S}'$ (provided $\mathrm{t} \notin \mathrm{Dom}(\mathcal{S}')$)	type component
	$\mid$	$\mathrm{x} : v, \mathcal{S}'$ (provided $\mathrm{x} \notin \mathrm{Dom}(\mathcal{S}')$)	value component
	$\mid$	$\mathrm{X} : \mathcal{M}, \mathcal{S}'$ (provided $\mathrm{X} \notin \mathrm{Dom}(\mathcal{S}')$)	*module component*
	$\mid$	$\epsilon_{\mathcal{S}}$	empty structure
$\mathcal{F} \in \mathit{Fun}$	$::=$	$\forall P.\mathcal{M} \to \mathcal{M}'$	*functor*
$\mathcal{M} \in \mathit{Mod}$	$::=$	$\mathcal{S}$	structure
	$\mid$	$\mathcal{F}$	functor
$\mathcal{X} \in \mathit{ExMod}$	$::=$	$\exists P.\mathcal{M}$	existential *module*
$\mathcal{L} \in \mathit{Sig}$	$::=$	$\Lambda P.\mathcal{M}$	signature

$$\mathcal{C} \in \mathit{Context} \quad \stackrel{\mathrm{def}}{=} \quad \left\{ C \cup \mathcal{C}_{\mathrm{t}} \cup \mathcal{C}_{\mathrm{x}} \cup \mathcal{C}_{\mathrm{X}} \;\middle|\; \begin{array}{l} C \in \mathit{CoreContext}, \\ \mathcal{C}_{\mathrm{t}} \in \mathrm{TypId} \stackrel{\mathrm{fin}}{\to} \mathit{Typ}, \\ \mathcal{C}_{\mathrm{x}} \in \mathrm{ValId} \stackrel{\mathrm{fin}}{\to} \mathit{ValTyp}, \\ \mathcal{C}_{\mathrm{X}} \in \mathrm{ModId} \stackrel{\mathrm{fin}}{\to} \mathit{Mod} \end{array} \right\}$$

Figure 5.16: Semantic Objects of Higher-Order Modules (cont.)

- A set of *types*, Typ^{κ}. A type $\tau^{\kappa} \in Typ^{\kappa}$ is either a type name of kind κ, a Core definable type of an equivalent Core kind, or a type *function* $\Lambda\alpha.\tau'$. In the case of a function, its kind κ must be a higher kind $\kappa' \to \kappa''$, its bound variable α must have the kind κ', and its body τ' must have the kind κ''.

In short, we have generalised the first-order concepts of Definition 3.4 by allowing higher kinds, well-kinded applications of type names to types, and well-kinded type functions. As in the first-order setting, type names enter Core semantic objects when they arise as the denotations of type occurrences.

Remark 5.4.1. Types, type names and type variables essentially define the β-normal terms of an extended, simply-typed λ-calculus. The base "terms" of the calculus are the Core definable types $d \in DefTyp$. The "base types" are the Core kinds $\mathrm{k} \in \mathrm{DefKind}$. Kinds $\kappa \in Kind$ play the role of "types" constructed with the "function space" $\kappa \to \kappa'$ from "base types". The motivations for restricting our attention to terms in β-normal form are twofold:

- We avoid the formulation of β-equivalence, and, more importantly, its extension to all other semantic objects of Higher-Order Modules.
- Terms enjoy the technically convenient property that if two terms are equivalent, then their sets of free variables are equivalent as well. Moreover, these are the "necessarily" free variables of the terms. This property simplifies the proofs in subsequent sections. Note that the property still holds when we identify terms up to η-equivalence.

The disadvantages of restricting ourselves to terms in β-normal form are also twofold:

- Their grammar is slightly more complex (two syntactic classes of type names and types, instead of just one). This complicates the definition of the usual operations on terms.
- The straightforward substitution of terms for variables does not respect the structure of terms. Instead, we need to define a more refined notion of substitution (i.e. realisation) that performs β-reduction on the fly.

Definition 5.2 (Equivalence of Types). Once again, using the Core operation η, every type name ν of Core kind k can be viewed as an equivalent Core definable type $\eta(\nu)$. We implicitly identify types that are equivalent up to such η-expansions and extend the operation η to operate on types of Core kind k (any k) as follows:

$$\begin{array}{rcl}\hat{\eta}^{\mathrm{k}}(\_) & \in & \mathit{Typ}^{\mathrm{k}} \rightarrow \mathit{DefTyp}^{\mathrm{k}} \\ \hat{\eta}^{\mathrm{k}}(\tau) & \overset{\mathrm{def}}{=} & \begin{cases} \eta(\nu) & \text{if } \tau \equiv \nu \in \mathit{TypNam}^{\mathrm{k}} \\ d & \text{if } \tau \equiv d \in \mathit{DefTyp}^{\mathrm{k}} \end{cases}\end{array}$$

From now on, we will identify any type of the form $\Lambda\alpha.\nu\ \alpha \in \mathit{TypNam}^{\kappa\rightarrow\kappa'}$ with its η-contraction ν, provided $\alpha \notin \mathrm{FV}(\nu)$ and $\nu \in \mathit{TypNam}^{\kappa\rightarrow\kappa'}$.

Moreover, we only consider as valid those equations between pairs of type variables, type names and definable types that compare objects of the same kind.

Definition 5.3 (Structures). A *semantic structure* $\mathcal{S} \in \mathit{Str}$ is a nested association list, binding identifiers to types (of Core kind), value types and semantic modules, i.e. both structures and functors. The *domain* of $\mathcal{S}$, written $\mathrm{Dom}(\mathcal{S})$, is the finite set of identifiers it binds:

$$\begin{array}{rcl}\mathrm{Dom}(\_) & \in & \mathit{Str} \rightarrow \mathrm{Fin}(\mathrm{TypId} \cup \mathrm{ValId} \cup \mathrm{ModId}) \\ \mathrm{Dom}(\epsilon_{\mathcal{S}}) & \overset{\mathrm{def}}{=} & \emptyset \\ \mathrm{Dom}(\mathrm{t} = \tau, \mathcal{S}) & \overset{\mathrm{def}}{=} & \{\mathrm{t}\} \cup \mathrm{Dom}(\mathcal{S}) \\ \mathrm{Dom}(\mathrm{x} : v, \mathcal{S}) & \overset{\mathrm{def}}{=} & \{\mathrm{x}\} \cup \mathrm{Dom}(\mathcal{S}) \\ \mathrm{Dom}(\mathrm{X} : \mathcal{M}, \mathcal{S}) & \overset{\mathrm{def}}{=} & \{\mathrm{X}\} \cup \mathrm{Dom}(\mathcal{S})\end{array}$$

The provisos on structures in Figure 5.16 ensure that identifiers are uniquely bound, allowing one to view a semantic structure as a triple of finite maps with corresponding (partial) retrieval functions:

$$\begin{array}{rcl}\_(\_) & \in & (\mathit{Str} \times \mathrm{TypId}) \rightharpoonup \mathit{Typ} \\ \epsilon_{\mathcal{S}}(\mathrm{t}) & \overset{\mathrm{def}}{=} & \text{undefined} \\ (\mathrm{t}' = \tau, \mathcal{S})(\mathrm{t}) & \overset{\mathrm{def}}{=} & \begin{cases} \tau & \text{if } \mathrm{t} = \mathrm{t}' \\ \mathcal{S}(\mathrm{t}) & \text{otherwise.} \end{cases} \\ (\mathrm{x} : v, \mathcal{S})(\mathrm{t}) & \overset{\mathrm{def}}{=} & \mathcal{S}(\mathrm{t}) \\ (\mathrm{X} : \mathcal{M}, \mathcal{S})(\mathrm{t}) & \overset{\mathrm{def}}{=} & \mathcal{S}(\mathrm{t})\end{array}$$

The retrieval functions for value and module bindings are defined similarly.

Definition 5.4 (Functors). A *semantic functor* $\mathcal{F} \in \mathit{Fun}$ is the type of a polymorphic function taking modules to modules. Consider a functor of type $\mathcal{F} \equiv \forall P.\mathcal{M} \to \mathcal{M}'$. Variables in P are bound simultaneously in $\mathcal{M}$ and $\mathcal{M}'$. These variables capture the type components of the domain $\mathcal{M}$ on which the functor behaves parametrically; their possible occurrence in the range $\mathcal{M}'$ caters for the propagation of type identities from the functor's actual argument to its result. In first-order Modules, the range of a functor is an existentially quantified semantic structure. Since we have chosen applicative functors as the more appropriate notion for Higher-Order Modules (recall our discussion in Section 5.2.2), the range $\mathcal{M}'$ is simply an unquantified semantic module.

Definition 5.5 (Signatures). Signature expressions denote *semantic signatures* $\mathcal{L} \in \mathit{Sig}$. The signature $\mathcal{L} \equiv \Lambda P.\mathcal{M}$ specifies a family of module types (either structures or functors), indexed by the realisation of type variables in P. Variables in P are bound in $\mathcal{M}$.

Definition 5.6 (Existential Modules). An *existential module* $\mathcal{X} \in \mathit{ExMod}$ is an existentially quantified module type of the form $\exists P.\mathcal{M}$. Variables in P are bound in $\mathcal{M}$. Intuitively, $\exists P.\mathcal{M}$ is the type of any abstracted module expression, whose actual type is *some* member of the family of types $\Lambda P.\mathcal{M}$.

Definition 5.7 (Contexts). A context $\mathcal{C} \in \mathit{Context}$ is a finite map assigning semantic objects to identifiers. Note that $\mathit{CoreContext} \subseteq \mathit{Context}$, so that every Core context is also a (Higher Order Modules) context, and that contexts support Core context operations. In addition to Core bindings, type, value, and module identifiers are mapped to types, value types, and modules respectively. Again, unlike semantic structures, contexts support re-bindings to identifiers. Subsequent bindings taking precedence over previous ones.

We will let $\mathrm{FV}(\mathcal{O})$ denote the set of variables free in $\mathcal{O}$, where the notions of free and bound are defined as usual. We identify semantic objects that are equivalent up to capture-avoiding, kind-preserving changes of bound variables.

Definition 5.8 (Realisations). As for first-order Modules, a *realisation* $\varphi \in \mathit{Real}$ is a kind-preserving finite map:

$$\varphi \in \mathit{Real} \overset{\text{def}}{=} \{f \in \mathit{TypVar} \overset{\text{fin}}{\to} \mathit{Typ} \mid \forall\kappa.\forall\alpha^{\kappa} \in \mathrm{Dom}(f).f(\alpha^{\kappa}) \in \mathit{Typ}^{\kappa}\},$$

defining a substitution on type variables. Unlike the realisations of first-order Modules, these realisations are *higher-order* substitutions.

Note that realisations are finite maps and we will often treat them as such. Moreover, to make it convenient to reason and state properties about the behaviour of realisations on semantic objects, we will define two auxiliary concepts. First, for a realisation φ, as in first-order Modules, we define its *region* as follows:

$$\begin{array}{rcl} \mathrm{Reg}(\_) & \in & \mathit{Real} \to \mathrm{Fin}(\mathit{TypVar}) \\ \mathrm{Reg}(\varphi) & \stackrel{\mathrm{def}}{=} & \displaystyle\bigcup_{\tau \in \mathrm{Rng}(\varphi)} \mathrm{FV}(\tau). \end{array}$$

Second, the set of variables *involved* in φ is captured by the definition:

$$\begin{array}{rcl} \mathrm{Inv}(\_) & \in & \mathit{Real} \to \mathrm{Fin}(\mathit{TypVar}) \\ \mathrm{Inv}(\varphi) & \stackrel{\mathrm{def}}{=} & \mathrm{Dom}(\varphi) \cup \mathrm{Reg}(\varphi). \end{array}$$

Definition 5.9 (Realisation of Type Names and Types). Realisation of type names and types is defined below. Because we essentially keep types in normal form, we may need to perform β-reductions on types and type names during realisation (see Remark 5.4.1):

$$\begin{array}{rcl} \_\,(\_) & \in & (\mathit{Real} \times \mathit{TypNam}^{\kappa}) \to \mathit{Typ}^{\kappa} \\ \varphi\,(\alpha) & \stackrel{\mathrm{def}}{=} & \begin{cases} \varphi(\alpha) & \text{if } \alpha \in \mathrm{Dom}(\varphi) \\ \alpha & \text{otherwise} \end{cases} \\ \varphi\,(\nu\ \tau) & \stackrel{\mathrm{def}}{=} & \begin{cases} \bar{\nu}\ (\varphi\,(\tau)) & \text{if } \varphi\,(\nu) \equiv \bar{\nu} \\ [\varphi\,(\tau)/\alpha]\,(\bar{\tau}) & \text{if } \varphi\,(\nu) \equiv \Lambda\alpha.\bar{\tau} \end{cases} \end{array}$$

and

$$\begin{array}{rcl} \_\,(\_) & \in & (\mathit{Real} \times \mathit{Typ}^{\kappa}) \to \mathit{Typ}^{\kappa} \\ \varphi\,(d) & \stackrel{\mathrm{def}}{=} & \varphi\,(d) \\ \varphi\,(\Lambda\alpha.\tau) & \stackrel{\mathrm{def}}{=} & \begin{cases} \Lambda\alpha.\varphi\,(\tau) & \text{if } \alpha \notin \mathrm{Inv}(\varphi) \\ \Lambda\beta.\varphi\,([\beta/\alpha]\,(\tau)) & \begin{array}{l}\text{if } \alpha \in \mathrm{Inv}(\varphi) \text{ and} \\ \beta \notin \mathrm{FV}(\Lambda\alpha.\tau) \cup \mathrm{Inv}(\varphi)\end{array} \end{cases} \\ \varphi\,(\nu) & \stackrel{\mathrm{def}}{=} & \varphi\,(\nu) \end{array}$$

Property 5.10 (Realisation is well-defined). *Since realisation is a combination of substitution and reduction, we should prove that it preserves kinds* *and terminates. This follows by translating kinds, types and type names into types and well-typed normal terms of the simply-typed λ-calculus. Realisation can be seen as type-preserving substitution, followed by standard β-reduction to β-normal form. By strong normalisation of the simply-typed λ-calculus with β-reduction, all such reduction sequences terminate. Hence realisation is total and well-defined.*

We extend realisations to structures, functors and modules, avoiding variable-capture by binding constructs ($\forall P.\mathcal{M} \to \mathcal{M}'$, $\exists P.\mathcal{M}$ and $\Lambda P.\mathcal{M}$) in the usual way.

Properties 5.11 (Realisations). *It is easy to verify that realisations enjoy the following properties. In later proofs, we shall frequently make implicit appeals to these properties.*

- *If* $\mathrm{Dom}(\varphi) \cap \mathrm{FV}(\mathcal{O}) = \emptyset$ *then* $\varphi\,(\mathcal{O}) = \mathcal{O}$.
- *If* $\mathrm{Dom}(\varphi) \cap \mathrm{FV}(\mathcal{O}) = \emptyset$ *then* $(\varphi \mid \varphi')\,(\mathcal{O}) = \varphi'\,(\mathcal{O})$.
- *If* $\mathrm{Dom}(\varphi) \cap \mathrm{Inv}(\varphi') = \emptyset$ *then* $(\varphi \mid \varphi')\,(\mathcal{O}) = \varphi\,(\varphi'\,(\mathcal{O}))$.
- *If* $\alpha \notin \mathrm{Inv}(\varphi)$ *then* $\varphi\,(\Lambda\alpha.\tau) = \Lambda\alpha.\varphi\,(\tau)$.
- *If* $\mathrm{Inv}(\varphi) \cap P = \emptyset$ *then*
  - $\varphi\,(\Lambda P.\mathcal{M}) = \Lambda P.\varphi\,(\mathcal{M})$,
  - $\varphi\,(\exists P.\mathcal{M}) = \exists P.\varphi\,(\mathcal{M})$, *and*
  - $\varphi\,(\forall P.\mathcal{M} \to \mathcal{M}') = \forall P.\varphi\,(\mathcal{M}) \to \varphi\,(\mathcal{M}')$.

Here, as in Definition 3.3, the operation $\varphi \mid \varphi'$ *defines a* parallel *realisation, i.e. provided* $\mathrm{Dom}(\varphi) \cap \mathrm{Dom}(\varphi') = \emptyset$ *then the* parallel *realisation* $\varphi \mid \varphi'$ *is the realisation* $\varphi \mid \varphi' \overset{\mathrm{def}}{=} \varphi \cup \varphi'$ *(viewing* φ *and* φ' *as sets) with domain* $\mathrm{Dom}(\varphi) \cup \mathrm{Dom}(\varphi')$.

5.4.1 Specifying Enrichment

In first-order Modules, the intuition motivating the enrichment relation is that, given a phrase of type $\mathcal{O}$, then provided $\mathcal{O} \succeq \mathcal{O}'$, we may also use the phrase as if it had the less general type $\mathcal{O}'$. The question is how to generalise this notion to the higher-order setting. In particular, what shall we mean when we say that one functor enriches another? Since functors are polymorphic, we might expect that a more polymorphic functor enriches any less polymorphic one that can be obtained from the richer functor by a realisation of its type parameters. This is similar to the way generalisation of Core-ML value types is defined. We could stop here, and proceed with this definition, but in fact, we will take it a little further by adapting the usual contra-variant definition of subtyping on function spaces: roughly speaking (and ignoring polymorphism for now), we shall also allow $\mathcal{F} \succeq \mathcal{F}'$ if every functor of type $\mathcal{F}$ may be applied to any argument in the domain of $\mathcal{F}'$, i.e. the domain of $\mathcal{F}$ is *at most* as rich as the domain of $\mathcal{F}'$, yielding a result *at least* as rich as the range of $\mathcal{F}'$, i.e. the range of $\mathcal{F}$ enriches the range of $\mathcal{F}'$. The difficulty lies in combining the two notions of polymorphic generalisation and contra-variant enrichment in a single definition. Intuitively, this idea is captured by the following specification:

Specification 5.12 (Enrichment). *We generalise the enrichment relation from (first-order) Modules, by extending its definition on structures to both functors and modules.*

$\boxed{\mathcal{S} \succeq \mathcal{S}'}$ *Given two semantic structures $\mathcal{S}$ and $\mathcal{S}'$, $\mathcal{S}$* enriches *$\mathcal{S}'$, written $\mathcal{S} \succeq \mathcal{S}'$, if, and only if, the following conditions hold.*

- $\mathrm{Dom}(\mathcal{S}) \supseteq \mathrm{Dom}(\mathcal{S}')$,
- *for each type identifier* $\mathrm{t} \in \mathrm{Dom}(\mathcal{S}')$, $\mathcal{S}(\mathrm{t}) = \mathcal{S}'(\mathrm{t})$,
- *for each value identifier* $\mathrm{x} \in \mathrm{Dom}(\mathcal{S}')$, $\mathcal{S}(\mathrm{x}) \succeq \mathcal{S}'(\mathrm{x})$,
- *for each module identifier* $\mathrm{X} \in \mathrm{Dom}(\mathcal{S}')$, $\mathcal{S}(\mathrm{X}) \succeq \mathcal{S}'(\mathrm{X})$.

$\boxed{\mathcal{F} \succeq \mathcal{F}'}$ *Given two functors $\mathcal{F}$ and $\mathcal{F}'$, $\mathcal{F}$* enriches *$\mathcal{F}'$, written $\mathcal{F} \succeq \mathcal{F}'$, if, and only if, every instance of $\mathcal{F}'$ is an instance of $\mathcal{F}$, i.e. for any modules $\mathcal{M}$ and $\mathcal{M}'$, $\mathcal{F}' > \mathcal{M} \to \mathcal{M}'$ implies $\mathcal{F} > \mathcal{M} \to \mathcal{M}'$.*

$\boxed{\mathcal{M} \succeq \mathcal{M}'}$ *Given two modules $\mathcal{M}$ and $\mathcal{M}'$, $\mathcal{M}$* enriches *$\mathcal{M}'$, written $\mathcal{M} \succeq \mathcal{M}'$, if, and only if,* either*:*

- *$\mathcal{M}$ and $\mathcal{M}'$ are both structures, i.e. $\mathcal{M} \equiv \mathcal{S}$ and $\mathcal{M}' \equiv \mathcal{S}'$, and $\mathcal{S} \succeq \mathcal{S}'$;* or
- *$\mathcal{M}$ and $\mathcal{M}'$ are both functors, i.e. $\mathcal{M} \equiv \mathcal{F}$ and $\mathcal{M}' \equiv \mathcal{F}'$, and $\mathcal{F} \succeq \mathcal{F}'$.*

$\boxed{\mathcal{F} > \mathcal{M} \to \mathcal{M}'}$ *A* functor instance *$\mathcal{M} \to \mathcal{M}'$ is the type of a monomorphic function on modules.*

Given functor $\mathcal{F} \equiv \forall P.\mathcal{M}_P \to \mathcal{M}'_P$, $\mathcal{M} \to \mathcal{M}'$ is an instance *of $\mathcal{F}$, written $\mathcal{F} > \mathcal{M} \to \mathcal{M}'$, if, and only if, for some realisation φ with $\mathrm{Dom}(\varphi) = P$, $\mathcal{M} \succeq \varphi(\mathcal{M}_P)$ and $\varphi(\mathcal{M}'_P) \succeq \mathcal{M}'$.*

Unfortunately, Specification 5.12 cannot be taken as a proper (inductive) definition of the enrichment relations, since one of the relations we are specifying occurs in the antecedent (i.e. in a negative position) of the clause relating functors. However, it is relatively easy for a programmer to understand, and can be treated as a specification of the enrichment relations we will define. In the next section, we will give a proper inductive definition of enrichment and show that it satisfies Specification 5.12. For the moment, we can observe that the property expressed by the specification is obviously reflexive and transitive, forming a good basis for a subtyping relation.

As in first-order Modules, matching a module against a signature is defined as a combination of realisation and enrichment:

Definition 5.13 (Signature Matching). A module $\mathcal{M}$ *matches* a signature $\mathcal{L} \equiv \Lambda P.\mathcal{M}'$ if, and only if, there exists a realisation φ such that $\mathcal{M} \succeq \varphi(\mathcal{M}')$ and $\mathrm{Dom}(\varphi) = P$.

5.4.2 Defining Enrichment

As we discussed, Specification 5.12, although intuitive, cannot serve as a definition of the enrichment relations between structures, functors and modules. In this section, we *define* enrichment as a collection of inductive relations and show that they form a pre-order that is closed under realisation. Using these properties, we can then prove that our definition satisfies Specification 5.12. Our proofs rely on certain assumptions about the Core language which we will state as hypotheses. These hypotheses must be proved separately for each instantiation of the Core language. But first, we give the definition:

Definition 5.14 (Enrichment). The enrichment relations between structures, functors and modules

$$\begin{array}{rcl} \_ \succeq \_ & \in & \mathit{Str} \times \mathit{Str} \\ \_ \succeq \_ & \in & \mathit{Fun} \times \mathit{Fun} \\ \_ \succeq \_ & \in & \mathit{Mod} \times \mathit{Mod} \end{array}$$

are defined as the least relations closed under the rules in Figure 5.17. Note that Rules ($\succeq$-3) and ($\succeq$-4) allow module enrichment to be derived from structure and functor enrichment.

Properties of Enrichment

We can now verify the properties of enrichment that justify its use as a subtyping relation.

Structure Enrichment $\boxed{\mathcal{S} \succeq \mathcal{S}'}$

$$\frac{\begin{array}{l}\mathrm{Dom}(\mathcal{S}) \supseteq \mathrm{Dom}(\mathcal{S}') \\ \forall \mathrm{t} \in \mathrm{Dom}(\mathcal{S}').\mathcal{S}(\mathrm{t}) = \mathcal{S}'(\mathrm{t}) \\ \forall \mathrm{x} \in \mathrm{Dom}(\mathcal{S}').\mathcal{S}(\mathrm{x}) \succeq \mathcal{S}'(\mathrm{x}) \\ \forall \mathrm{X} \in \mathrm{Dom}(\mathcal{S}').\mathcal{S}(\mathrm{X}) \succeq \mathcal{S}'(\mathrm{X})\end{array}}{\mathcal{S} \succeq \mathcal{S}'} \qquad (\succeq\text{-1})$$

Functor Enrichment $\boxed{\mathcal{F} \succeq \mathcal{F}'}$

$$\frac{\begin{array}{ll}\mathcal{M}_Q \succeq \varphi(\mathcal{M}_P) & \varphi(\mathcal{M}'_P) \succeq \mathcal{M}'_Q \\ \mathrm{Dom}(\varphi) = P & Q \cap \mathrm{FV}(\forall P.\mathcal{M}_P \to \mathcal{M}'_P) = \emptyset\end{array}}{\forall P.\mathcal{M}_P \to \mathcal{M}'_P \succeq \forall Q.\mathcal{M}_Q \to \mathcal{M}'_Q} \qquad (\succeq\text{-2})$$

Module Enrichment $\boxed{\mathcal{M} \succeq \mathcal{M}'}$

$$\frac{\mathcal{S} \succeq \mathcal{S}'}{\mathcal{S} \succeq \mathcal{S}'} \qquad (\succeq\text{-3})$$

$$\frac{\mathcal{F} \succeq \mathcal{F}'}{\mathcal{F} \succeq \mathcal{F}'} \qquad (\succeq\text{-4})$$

Figure 5.17: An inductive definition of enrichment for Higher-Order Modules.

We need to assume that the Core satisfies the following hypothesis:

Hypothesis 5.15 (Reflexivity of Value Type Enrichment).

$$v \succeq v.$$

Then it is easy to prove:

Property 5.16 (Reflexivity).

$$\mathcal{O} \succeq \mathcal{O}.$$

Proof. *The proof is an easy structural induction on $\mathcal{O}$, with an appeal to Hypothesis 5.15.*

Moreover, we need to know that enrichment between value types is closed under realisation of type variables:

Hypothesis 5.17 (Closure under Realisation of Value Type Enrichment).

$$v \succeq v' \supset \varphi\,(v) \succeq \varphi\,(v').$$

This hypothesis allows us to prove a similar closure property for the enrichment relation on structures, functors and modules:

Lemma 5.18 (Closure under Realisation of Enrichment).

$$\mathcal{O} \succeq \mathcal{O}' \supset \varphi\,(\mathcal{O}) \succeq \varphi\,(\mathcal{O}').$$

Proof. *Because we need to be able to apply the induction hypothesis to a modified realisation in case ($\succeq$ -2), we actually need to prove the statement:*

$$\mathcal{O} \succeq \mathcal{O}' \supset \forall\varphi.\varphi\,(\mathcal{O}) \succeq \varphi\,(\mathcal{O}'),$$

that quantifies over all φ. The proof then follows by induction on the rules defining enrichment. Note that we need to appeal to Hypothesis 5.17 in case $\succeq$ -1.

(As an aside, we point out that the strategy of first choosing some arbitrary, but fixed φ, and then trying to prove

$$\mathcal{O} \succeq \mathcal{O}' \supset \varphi\,(\mathcal{O}) \succeq \varphi\,(\mathcal{O}')$$

directly by induction on the rules defining enrichment, fails, although it works fine for the enrichment relation of first-order Modules (Definition 3.17).)

Finally, we shall need to assume that Core enrichment is transitive.

Hypothesis 5.19 (Transitivity of Value Type Enrichment).

$$v \succeq v' \wedge v' \succeq v'' \supset v \succeq v''.$$

We can then prove that the enrichment relations on structures, functors and modules is transitive.

Property 5.20 (Transitivity).

$$\mathcal{O} \succeq \mathcal{O}' \supset \mathcal{O}' \succeq \mathcal{O}'' \supset \mathcal{O} \succeq \mathcal{O}''.$$

Proof. *The proof is a little tricky because of the simultaneous use of contravariance and realisation in Rule* ($\succeq$ -2)*. The trick is to use Lemma 5.18 to prove the stronger property:*

$$\mathcal{O} \succeq \mathcal{O}' \supset \left(\begin{array}{cc} & \forall\varphi, \mathcal{O}''.\varphi(\mathcal{O}') \succeq \mathcal{O}'' \supset \varphi(\mathcal{O}) \succeq \mathcal{O}'' \\ \wedge & \forall\varphi, \mathcal{O}''.\mathcal{O}'' \succeq \varphi(\mathcal{O}) \supset \mathcal{O}'' \succeq \varphi(\mathcal{O}') \end{array} \right)$$

which succumbs to rule induction and an appeal to Hypothesis 5.19. The stronger induction hypothesis corresponds to splitting the proof of transitivity into a simultaneous proof of transitivity for objects enriched by realisations of $\mathcal{O}'$, together with a proof of transitivity for objects enriching realisations of $\mathcal{O}$.

Lemma 5.20 follows easily from the first conjunct, by choosing the empty, or identity, realisation.

(As an aside, we point out that the simpler strategy of trying to prove

$$\mathcal{O} \succeq \mathcal{O}' \supset \forall\mathcal{O}''.\mathcal{O}' \succeq \mathcal{O}'' \supset \mathcal{O} \succeq \mathcal{O}''$$

directly by induction on the rules defining enrichment, fails, although it works fine for the enrichment relation of first-order Modules (Definition 3.17).)

We now have everything we need to prove that our enrichment relations satisfy Specification 5.12. We first define, as in Specification 5.12:

Definition 5.21 (Functor Instance). A *functor instance* $\mathcal{M} \to \mathcal{M}'$ is the type of a monomorphic function on modules.

Given functor $\mathcal{F} \equiv \forall P.\mathcal{M}_P \to \mathcal{M}'_P$, $\mathcal{M} \to \mathcal{M}'$ is an *instance* of $\mathcal{F}$, written $\mathcal{F} > \mathcal{M} \to \mathcal{M}'$, if, and only if, for some realisation φ with $\mathrm{Dom}(\varphi) = P$, $\mathcal{M} \succeq \varphi(\mathcal{M}_P)$ and $\varphi(\mathcal{M}'_P) \succeq \mathcal{M}'$.

We will need the following, simple observation:

Observation 5.22 (Generic Instance). *Every functor is a generic instance of itself:*

$$\forall P.\mathcal{M} \to \mathcal{M}' > \mathcal{M} \to \mathcal{M}'$$

Proof. *Easy, by choosing φ to be the identity on P and appealing to Property 5.16.*

The key to proving that our relations satisfy their specification is to show the following lemma:

Lemma 5.23 (Characterisation).

$$\mathcal{F} \succeq \mathcal{F}' \text{ if, and only if, } \forall \mathcal{M}, \mathcal{M}'.\mathcal{F}' > \mathcal{M} \to \mathcal{M}' \supset \mathcal{F} > \mathcal{M} \to \mathcal{M}'.$$

Proof. *The reverse direction is easy and relies on Observation 5.22. The forward direction is harder and requires appeals to Property 5.20 and Lemma 5.18.*

It is now straightforward to verify:

Theorem 5.24 (Satisfaction). *The family of relations $\_ \succeq \_$ satisfies Specification 5.12.*

Proof. *Easy using Lemma 5.23.*

5.5 Static Semantics

The static semantics of Modules is defined by the judgements in Figure 5.18. We have indicated, below each judgement, its intended English reading. The judgements are defined by the following rules:

Denotation Rules

The denotation rules for signature bodies and signature expressions are the same as their first-order counterparts (cf. Section 3.1.3), except that we have adopted the notation $\Lambda P.\mathcal{M}$ instead of $(P)\mathcal{M}$ for semantic signatures. Rule (H-4) generalises the first-order Rule (E-4) by catering for module specifications instead of structure specifications, but is otherwise unchanged. Rule (H-7) is new and deals with functor signatures, incorporating the ideas we sketched in Section 5.2.1. Finally, the original denotation rule for projecting a type component from a path (Rule (E-8)) has been generalised to support the projection of a type component from an arbitrary module expression (Rule (H-9)).

$\mathcal{C} \vdash \mathrm{S} \triangleright \mathcal{L}$

In context $\mathcal{C}$, signature expression S denotes signature $\mathcal{L}$.

$\mathcal{C} \vdash \mathrm{B} \triangleright \mathcal{L}$

In context $\mathcal{C}$, signature body B denotes signature $\mathcal{L}$.

$\mathcal{C} \vdash \mathrm{do} \triangleright d$

In context $\mathcal{C}$, type occurrence do denotes definable type d.

(a) Denotation Judgements

$\mathcal{C} \vdash \mathrm{b} : \mathcal{X}$

In context $\mathcal{C}$, structure body b has existential module type $\mathcal{X}$.

$\mathcal{C} \vdash \mathrm{m} : \mathcal{X}$

In context $\mathcal{C}$, module expression m has existential module type $\mathcal{X}$.

$\mathcal{C} \vdash \mathrm{vo} : v$

In context $\mathcal{C}$, value occurrence vo has value type v.

(b) Classification Judgements

Figure 5.18: Higher-Order Modules Judgements

Signature Bodies $\boxed{\mathcal{C} \vdash \mathrm{B} \triangleright \mathcal{L}}$

$$\frac{\begin{array}{ll}\mathcal{C} \vdash \mathrm{d} \triangleright d & P \cap \mathrm{FV}(d) = \emptyset \\ \mathcal{C}[\mathrm{t} = d] \vdash \mathrm{B} \triangleright \Lambda P.\mathcal{S} & \mathrm{t} \notin \mathrm{Dom}(\mathcal{S})\end{array}}{\mathcal{C} \vdash \textbf{type}\ \mathrm{t} = \mathrm{d}; \mathrm{B} \triangleright \Lambda P.\mathrm{t} = d, \mathcal{S}} \quad \text{(H-1)}$$

$$\frac{\mathcal{C}[\mathrm{t} = \alpha^{\mathrm{k}}] \vdash \mathrm{B} \triangleright \Lambda P.\mathcal{S} \quad \alpha^{\mathrm{k}} \notin \mathrm{FV}(\mathcal{C}) \cup P \quad \mathrm{t} \notin \mathrm{Dom}(\mathcal{S})}{\mathcal{C} \vdash \textbf{type}\ \mathrm{t} : \mathrm{k}; \mathrm{B} \triangleright \Lambda\{\alpha^{\mathrm{k}}\} \cup P.\mathrm{t} = \alpha^{\mathrm{k}}, \mathcal{S}} \quad \text{(H-2)}$$

$$\frac{\begin{array}{ll}\mathcal{C} \vdash \mathrm{v} \triangleright v & P \cap \mathrm{FV}(v) = \emptyset \\ \mathcal{C}[\mathrm{x} : v] \vdash \mathrm{B} \triangleright \Lambda P.\mathcal{S} & \mathrm{x} \notin \mathrm{Dom}(\mathcal{S})\end{array}}{\mathcal{C} \vdash \textbf{val}\ \mathrm{x} : \mathrm{v}; \mathrm{B} \triangleright \Lambda P.\mathrm{x} : v, \mathcal{S}} \quad \text{(H-3)}$$

$$\frac{\begin{array}{ll}\mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{M} & \\ \mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash \mathrm{B} \triangleright \Lambda Q.\mathcal{S} & P \cap \mathrm{FV}(\mathcal{C}) = \emptyset \\ Q \cap (P \cup \mathrm{FV}(\mathcal{M})) = \emptyset & \mathrm{X} \notin \mathrm{Dom}(\mathcal{S})\end{array}}{\mathcal{C} \vdash \textbf{module}\ \mathrm{X} : \mathrm{S}; \mathrm{B} \triangleright \Lambda P \cup Q.\mathrm{X} : \mathcal{M}, \mathcal{S}} \quad \text{(H-4)}$$

$$\frac{}{\mathcal{C} \vdash \epsilon_{\mathrm{B}} \triangleright \Lambda\emptyset.\epsilon_{\mathcal{S}}} \quad \text{(H-5)}$$

Signature Expressions $\boxed{\mathcal{C} \vdash \mathrm{S} \triangleright \mathcal{L}}$

$$\frac{\mathcal{C} \vdash \mathrm{B} \triangleright \mathcal{L}}{\mathcal{C} \vdash \textbf{sig}\ \mathrm{B}\ \textbf{end} \triangleright \mathcal{L}} \tag{H-6}$$

$$\frac{\begin{array}{l}\mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{M} \\ P \cap \mathrm{FV}(\mathcal{C}) = \emptyset \quad P = \{\alpha_0^{\kappa_0}, \ldots, \alpha_{n-1}^{\kappa_{n-1}}\} \\ \mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash \mathrm{S}' \triangleright \Lambda Q.\mathcal{M}' \\ Q' \cap (P \cup \mathrm{FV}(\mathcal{M}) \cup \mathrm{FV}(\Lambda Q.\mathcal{M}')) = \emptyset \\ [Q'/Q] = \{\beta^\kappa \mapsto \beta^{\kappa_0 \to \cdots \kappa_{n-1} \to \kappa}\ \alpha_0 \cdots \alpha_{n-1} | \beta^\kappa \in Q\} \\ Q' = \{\beta^{\kappa_0 \to \cdots \kappa_{n-1} \to \kappa} | \beta^\kappa \in Q\}\end{array}}{\mathcal{C} \vdash \textbf{funsig}(\mathrm{X}{:}\mathrm{S})\mathrm{S}' \triangleright \Lambda Q'.\forall P.\mathcal{M} \to [Q'/Q]\,(\mathcal{M}')} \tag{H-7}$$

(H-7) The range signature S' denotes a semantic signature $\Lambda Q.\mathcal{M}'$. We want to capture the fact that types specified in S' may have a functional dependency on the type parameters in P. We cater for these dependencies by applying the realisation $[Q'/Q]$ to $\mathcal{M}'$. This effectively parameterises each occurrence in $\mathcal{M}'$ of a variable $\beta \in Q$ by the variables P. The kinds of the variables in Q must be adjusted to reflect this, yielding the set Q'. Having modified variables in Q to take account of their implicit dependencies on P, we can move the parameterisation over Q from the range, i.e. $\Lambda Q.\mathcal{M}'$, to a position *outside* the entire functor yielding the signature $\Lambda Q'.\forall P.\mathcal{M} \to [Q'/Q]\,(\mathcal{M}')$. This is how we systematically treat type parameters arising from the range of a functor signature. Note that, even though these parameters are bound by an "outermost" Λ, they still manage to encode the dependency of the functor's result types on the functor's type arguments.

Type Occurrences $\boxed{\mathcal{C} \vdash \mathrm{do} \triangleright d}$

$$\frac{\mathrm{t} \in \mathrm{Dom}(\mathcal{C}) \quad \mathcal{C}(\mathrm{t}) = \tau}{\mathcal{C} \vdash \mathrm{t} \triangleright \hat{\eta}(\tau)} \tag{H-8}$$

$$\frac{\mathcal{C} \vdash \mathrm{m} : \exists P.\mathcal{S} \quad \mathrm{t} \in \mathrm{Dom}(\mathcal{S}) \quad \mathcal{S}(\mathrm{t}) = \tau \quad P \cap \mathrm{FV}(\tau) = \emptyset}{\mathcal{C} \vdash \mathrm{m.t} \triangleright \hat{\eta}(\tau)} \tag{H-9}$$

(H-9) The side condition $P \cap \mathrm{FV}(\tau) = \emptyset$ ensures that existential variables in P do not escape their scope. Note that m must be a structure, not a functor.

Classification Rules

The classification rules for structure bodies are the same as their first-order counterparts (cf. Section 4.2.1): Rules (H-12) and (H-13) merely generalise the first-order Rules (T-15) and (T-16) by catering for module definitions instead of structure definitions, but are otherwise unchanged.

The classification rules for module expressions deserve the most comment. Rules (H-15) and (H-16) subsume the role of the first-order Rules (T-19), (E-9) and (E-10), but also cater for generalised module projections. Rules (H-18) and (H-19) are new and formalise the applicative semantics of Section 5.2.2 for anonymous functors and module applications. The remaining rules for encapsulating a structure body, curtailing a module by a signature and abstracting a module by a signature are unchanged from the first-order rules of Section 4.2.1, except that the last two now apply to both structures and functors.

Finally, the original classification rule for projecting a value component from a path (Rule (E-12)) has been generalised to support the projection of a value component from an arbitrary module expression (Rule (H-23)).

This static semantics, as it stands, does not yield a type checking *algorithm*. For instance, in Rule (H-19), classifying a functor application, we have to "guess" a realisation φ such that $\mathcal{M}'' \succeq \varphi(\mathcal{M}')$ and $\mathrm{Dom}(\varphi) = Q$. In Section 5.6, we define an algorithm that finds a suitable matching realisation provided it exists. We can then replace the problematic premise $\mathcal{M}'' \succeq \varphi(\mathcal{M}')$ with an appeal to this algorithm. Similar comments apply to the rules for module curtailment, Rule H-20, and module abstraction, Rule H-21. All the other rules, by contrast, are syntax directed and can be used directly to define a type checking algorithm.

Structure Bodies $\boxed{\mathcal{C} \vdash \mathrm{b} : \mathcal{X}}$

$$\frac{\begin{array}{ll}\mathcal{C} \vdash \mathrm{d} \rhd d & P \cap \mathrm{FV}(d) = \emptyset \\ \mathcal{C}[\mathrm{t} = d] \vdash \mathrm{b} : \exists P.\mathcal{S} & \mathrm{t} \notin \mathrm{Dom}(\mathcal{S})\end{array}}{\mathcal{C} \vdash \mathbf{type}\ \mathrm{t} = \mathrm{d}; \mathrm{b} : \exists P.\mathrm{t} = d, \mathcal{S}} \quad \text{(H-10)}$$

$$\frac{\begin{array}{ll}\mathcal{C} \vdash \mathrm{e} : v & P \cap \mathrm{FV}(v) = \emptyset \\ \mathcal{C}[\mathrm{x} : v] \vdash \mathrm{b} : \exists P.\mathcal{S} & \mathrm{x} \notin \mathrm{Dom}(\mathcal{S})\end{array}}{\mathcal{C} \vdash \mathbf{val}\ \mathrm{x} = \mathrm{e}; \mathrm{b} : \exists P.\mathrm{x} : v, \mathcal{S}} \quad \text{(H-11)}$$

$$\frac{\begin{array}{ll}\mathcal{C} \vdash \mathrm{m} : \exists P.\mathcal{M} & P \cap \mathrm{FV}(\mathcal{C}) = \emptyset \\ \mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash \mathrm{b} : \exists P'.\mathcal{S} & \\ P' \cap (P \cup \mathrm{FV}(\mathcal{M})) = \emptyset & \mathrm{X} \notin \mathrm{Dom}(\mathcal{S})\end{array}}{\mathcal{C} \vdash \textbf{module}\ \mathrm{X} = \mathrm{m}; \mathrm{b} : \exists P \cup P'.\mathrm{X} : \mathcal{M}, \mathcal{S}} \tag{H-12}$$

$$\frac{\begin{array}{ll}\mathcal{C} \vdash \mathrm{m} : \exists P.\mathcal{M} & \mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash \mathrm{b} : \exists P'.\mathcal{S} \\ P \cap \mathrm{FV}(\mathcal{C}) = \emptyset & P' \cap P = \emptyset\end{array}}{\mathcal{C} \vdash \textbf{local}\ \mathrm{X}\ =\ \mathrm{m}\ \textbf{in}\ \mathrm{b} : \exists P \cup P'.\mathcal{S}} \tag{H-13}$$

$$\frac{}{\mathcal{C} \vdash \epsilon_{\mathrm{b}} : \exists \emptyset.\epsilon_{\mathcal{S}}} \tag{H-14}$$

Module Expressions $\boxed{\mathcal{C} \vdash \mathrm{m} : \mathcal{X}}$

$$\frac{\mathrm{X} \in \mathrm{Dom}(\mathcal{C}) \quad \mathcal{C}(\mathrm{X}) = \mathcal{M}}{\mathcal{C} \vdash \mathrm{X} : \exists \emptyset.\mathcal{M}} \tag{H-15}$$

$$\frac{\mathcal{C} \vdash \mathrm{m} : \exists P.\mathcal{S} \quad \mathrm{X} \in \mathrm{Dom}(\mathcal{S}) \quad \mathcal{S}(\mathrm{X}) = \mathcal{M}}{\mathcal{C} \vdash \mathrm{m.X} : \exists P.\mathcal{M}} \tag{H-16}$$

(H-16) As we have generalised the dot notation to apply to arbitrary module expressions, which may have existentially quantified types, we need to ensure that the projection of a sub-module X from an existential structure $\exists P.\mathcal{S}$ does not allow variables in P to escape their scope. Existentially quantifying over P in the result $\exists P.\mathcal{M}$ is sufficient. Moreover, this allows us to access deeply nested type or value components even though the intermediate sub-modules might contain existentially quantified variables. Note that the side-conditions of Rules (H-9) and (H-23) will prevent these variables from escaping their scope via projected type and value components. Also note that m must be a structure, not a functor.

$$\frac{\mathcal{C} \vdash \mathrm{b} : \mathcal{X}}{\mathcal{C} \vdash \textbf{struct}\ \mathrm{b}\ \textbf{end} : \mathcal{X}} \tag{H-17}$$

$$\frac{\begin{array}{l}\mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{M} \\ P \cap \mathrm{FV}(\mathcal{C}) = \emptyset \quad P = \{\alpha_0^{\kappa_0}, \ldots, \alpha_{n-1}^{\kappa_{n-1}}\} \\ \mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash \mathrm{m} : \exists Q.\mathcal{M}' \\ Q' \cap (P \cup \mathrm{FV}(\mathcal{M}) \cup \mathrm{FV}(\exists Q.\mathcal{M}')) = \emptyset \\ [Q'/Q] = \{\beta^\kappa \mapsto \beta^{\kappa_0 \to \cdots \kappa_{n-1} \to \kappa}\ \alpha_0 \cdots \alpha_{n-1} | \beta^\kappa \in Q\} \\ Q' = \{\beta^{\kappa_0 \to \cdots \kappa_{n-1} \to \kappa} | \beta^\kappa \in Q\}\end{array}}{\mathcal{C} \vdash \mathbf{functor}(\mathrm{X} : \mathrm{S})\mathrm{m} : \exists Q'.\forall P.\mathcal{M} \to [Q'/Q]\,(\mathcal{M}')} \tag{H-18}$$

(H-18) Note the applicative semantics of functors. Classifying the functor body m introduces existential types Q. In general, because m is classified in the extended context $\mathcal{C}[\mathrm{X} : \mathcal{M}]$, variables in Q may have hidden functional dependencies on the type parameters P of the formal argument X. We make these dependencies explicit by applying the realisation $[Q'/Q]$ to $\mathcal{M}'$. This effectively *skolemises* each occurrence in $\mathcal{M}'$ of a variable $\beta \in Q$ by the variables P. The kinds of the variables in Q must be adjusted to reflect this, yielding the set Q'. Having "raised" variables in Q by their implicit parameters, we can move the existential quantification over the functor range, i.e. $\exists Q.\mathcal{M}'$, to a position *outside* the entire functor yielding the existential module $\exists Q'.\forall P.\mathcal{M} \to [Q'/Q]\,(\mathcal{M}')$. This is how we systematically avoid the need to existentially quantify the range of a semantic functor.

$$\frac{\begin{array}{l}\mathcal{C} \vdash \mathrm{m} : \exists P.\forall Q.\mathcal{M}' \to \mathcal{M} \\ \mathcal{C} \vdash \mathrm{m}' : \exists P'.\mathcal{M}'' \\ P \cap (P' \cup \mathrm{FV}(\mathcal{M}'')) = \emptyset \\ P' \cap \mathrm{FV}(\forall Q.\mathcal{M}' \to \mathcal{M}) = \emptyset \\ \mathcal{M}'' \succeq \varphi\,(\mathcal{M}') \\ \mathrm{Dom}(\varphi) = Q\end{array}}{\mathcal{C} \vdash \mathrm{m}\ \mathrm{m}' : \exists P \cup P'.\varphi\,(\mathcal{M})} \tag{H-19}$$

(H-19) Note that, because the functor m is an anonymous module, its type may be existentially quantified (for instance, m might be an abstracted functor). However, because functors are applicative, the range of the functor will simply be a semantic module $\mathcal{M}$, not an existentially quantified type as in the first-order semantics. To classify the application, we first eliminate the existential quantifiers in both the type of functor and the type of the argument, yielding the semantic functor $\forall Q.\mathcal{M}' \to \mathcal{M}$, and semantic module $\mathcal{M}''$. We now choose a realisation

φ of the functor's type parameters Q such that $\mathcal{M}''$ enriches the realised domain $\varphi(\mathcal{M}')$. We then propagate this realisation through to the range $\mathcal{M}$ of the functor yielding the result type $\varphi(\mathcal{M})$. (Another way of saying the last two sentences is that we choose the functor instance $\forall Q.\mathcal{M}' \to \mathcal{M} > \mathcal{M}'' \to \varphi(\mathcal{M})$ appropriate to the domain $\mathcal{M}''$.) However, because the type $\varphi(\mathcal{M})$ may mention the eliminated existential variables P and P', we need to ensure that they cannot escape their scope. So we re-introduce an existential quantifier that hides both P and P' in the final type of the application $\exists P \cup P'.\varphi(\mathcal{M})$.

$$\frac{\begin{array}{ll} \mathcal{C} \vdash \mathrm{m} : \exists P.\mathcal{M} & \\ \mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P'.\mathcal{M}' & P \cap \mathrm{FV}(\Lambda P'.\mathcal{M}') = \emptyset \\ \mathcal{M} \succeq \varphi(\mathcal{M}') & \mathrm{Dom}(\varphi) = P' \end{array}}{\mathcal{C} \vdash \mathrm{m} \succeq \mathrm{S} : \exists P.\varphi(\mathcal{M}')} \tag{H-20}$$

(H-20) Applying the realisation φ to $\mathcal{M}'$ in the result $\exists P.\varphi(\mathcal{M}')$ ensures that the actual realisations of type components merely specified in S are retained. In particular, if m is a functor, then the curtailment preserves the actual argument-result dependencies of m that are merely specified, but not defined, in S.

$$\frac{\begin{array}{ll} \mathcal{C} \vdash \mathrm{m} : \exists P.\mathcal{M} & \\ \mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P'.\mathcal{M}' & P \cap \mathrm{FV}(\Lambda P'.\mathcal{M}') = \emptyset \\ \mathcal{M} \succeq \varphi(\mathcal{M}') & \mathrm{Dom}(\varphi) = P' \end{array}}{\mathcal{C} \vdash \mathrm{m} \setminus \mathrm{S} : \exists P'.\mathcal{M}'} \tag{H-21}$$

(H-21) As in Rule H-20 we require that there be *some* realisation φ such that $\mathcal{M}$ matches $\Lambda P'.\mathcal{M}'$. However, the type of $\mathrm{m} \setminus \mathrm{S}$ is $\exists P'.\mathcal{M}'$, not $\exists P.\varphi(\mathcal{M}')$. As a result, types merely specified in S are made abstract. In particular, if m is a functor, then the abstraction makes the actual argument-result dependencies of m that are merely specified, but not defined, in S abstract. Note that this is the only rule that introduces existentially quantified variables.

Value Occurrences $\boxed{\mathcal{C} \vdash \mathrm{vo} : v}$

$$\frac{\mathrm{x} \in \mathrm{Dom}(\mathcal{C}) \quad \mathcal{C}(\mathrm{x}) = v}{\mathcal{C} \vdash \mathrm{x} : v} \tag{H-22}$$

$$\frac{\mathcal{C} \vdash \mathrm{m} : \exists P.\mathcal{S} \quad \mathrm{x} \in \mathrm{Dom}(\mathcal{S}) \quad \mathcal{S}(\mathrm{x}) = v \quad P \cap \mathrm{FV}(v) = \emptyset}{\mathcal{C} \vdash \mathrm{m.x} : v} \tag{H-23}$$

(H-23) The side condition $P \cap \mathrm{FV}(v) = \emptyset$ ensures that existential variables in P do not escape their scope. Note that m must be a structure, not a functor.

5.6 An Algorithm for Matching

The rules defining the static semantics of Higher-Order Modules almost capture a type checking algorithm that, given a context and phrase, calculates the semantic object (if any) that the phrase denotes or is classified by. However, Rules H-19, H-20 and H-21 present a problem. Each of the rules carries premises that require the choice of an appropriate realisation such that one module enriches the realisation of another. We have not yet shown how such a realisation may be found. For (first-order) Modules it is easy to see that we can factor the process of matching a structure $\mathcal{S}$ to a signature $\Lambda P.\mathcal{S}'$ into two steps: first, we find an appropriate realisation φ, then we check that $\mathcal{S} \succeq \varphi(\mathcal{S}')$. Unfortunately, in the higher-order case, these two steps can no longer be carried out separately. In Rule $\succeq$-2, defining when one functor enriches another, we need to guess a realisation such that the domain of the less general functor enriches the realisation of the more general functor's domain. The notions of enrichment and matching are now intertwined, and it should come as no surprise that the algorithm for matching must simultaneously construct a realisation while verifying enrichment. Note that the realisations we need to construct are essentially higher-order substitutions, since semantic types define the terms of a simply typed λ-calculus (cf. Remark 5.4.1). Although, in general, higher-order unification of typed λ-terms is undecidable and non-unitary (i.e. there may be more than one solution to a given problem) [Hue75], we shall find that the matching problems encountered during type checking belong to a restricted class of problem for which matching is both decidable and unitary. Indeed, we will only need to construct realisations for type variables that occur in types that take the restricted form of *higher-order patterns*. The class of higher-order patterns was originally identified and studied by Miller [Mil91].

Definition 5.25 (Signature Matching Algorithm). Figure 5.19 defines the rules of an algorithm, $\forall P.\forall R \vdash \mathcal{O} \succeq \mathcal{O}' \downarrow \varphi$, that, given as inputs two sets of type variables P and R, and a pair of semantic objects $\mathcal{O}$ and $\mathcal{O}'$, be they structures, functors or modules, computes a realisation φ as its output, provided it succeeds.

The intention is that, provided certain conditions on its inputs hold, we

Structure Matching $\boxed{\forall P.\forall R \vdash \mathcal{S} \succeq \mathcal{S}' \downarrow \varphi}$

$$\frac{}{\forall P.\forall R \vdash \mathcal{S} \succeq \epsilon_{\mathcal{S}} \downarrow \emptyset} \quad (M\text{-}1)$$

$$\frac{\mathrm{t} \in \mathrm{Dom}(\mathcal{S}) \quad \mathcal{S}(\mathrm{t}) = \tau \quad \forall P.\forall R \vdash \mathcal{S} \succeq \mathcal{S}' \downarrow \varphi}{\forall P.\forall R \vdash \mathcal{S} \succeq \mathrm{t} = \tau, \mathcal{S}' \downarrow \varphi} \quad (M\text{-}2)$$

$$\frac{\begin{array}{l} \alpha \notin P \cup R \\ \mathrm{t} \in \mathrm{Dom}(\mathcal{S}) \\ \mathcal{S}(\mathrm{t}), (\alpha\ \beta_0 \cdots \beta_{n-1}) \in \mathit{Typ}^{\kappa} \\ \mathrm{FV}(\mathcal{S}(\mathrm{t})) \cap R \subseteq \{\beta_i \mid i \in [n]\} \\ \forall P.\forall R \vdash \mathcal{S} \succeq [\Lambda\beta_0 \cdots \beta_{n-1}.\mathcal{S}(\mathrm{t})/\alpha]\,(\mathcal{S}') \downarrow \varphi \end{array}}{\forall P.\forall R \vdash \mathcal{S} \succeq \mathrm{t} = \alpha\ \beta_0 \cdots \beta_{n-1}, \mathcal{S}' \downarrow ([\Lambda\beta_0 \cdots \beta_{n-1}.\mathcal{S}(\mathrm{t})/\alpha] \mid \varphi)} \quad (M\text{-}3)$$

$$\frac{\mathrm{x} \in \mathrm{Dom}(\mathcal{S}) \quad \mathcal{S}(\mathrm{x}) \succeq v \quad \forall P.\forall R \vdash \mathcal{S} \succeq \mathcal{S}' \downarrow \varphi}{\forall P.\forall R \vdash \mathcal{S} \succeq \mathrm{x} : v, \mathcal{S}' \downarrow \varphi} \quad (M\text{-}4)$$

$$\frac{\mathrm{X} \in \mathrm{Dom}(\mathcal{S}) \quad \forall P.\forall R \vdash \mathcal{S}(\mathrm{X}) \succeq \mathcal{M} \downarrow \varphi \quad \forall P.\forall R \vdash \mathcal{S} \succeq \varphi\,(\mathcal{S}') \downarrow \varphi'}{\forall P.\forall R \vdash \mathcal{S} \succeq \mathrm{X} : \mathcal{M}, \mathcal{S}' \downarrow (\varphi \mid \varphi')} \quad (M\text{-}5)$$

Functor Matching $\boxed{\forall P.\forall R \vdash \mathcal{F} \succeq \mathcal{F}' \downarrow \varphi}$

$$\frac{\begin{array}{l} N \cap (P \cup R \cup M) = \emptyset \\ M \cap (P \cup R) = \emptyset \\ \forall P \cup R \cup M.\forall\emptyset \vdash \mathcal{M}_M \succeq \mathcal{M}_N \downarrow \varphi' \\ \forall P.\forall R \cup M \vdash \varphi'\,(\mathcal{M}'_N) \succeq \mathcal{M}'_M \downarrow \varphi \end{array}}{\forall P.\forall R \vdash \forall N.\mathcal{M}_N \to \mathcal{M}'_N \succeq \forall M.\mathcal{M}_M \to \mathcal{M}'_M \downarrow \varphi} \quad (M\text{-}6)$$

Module Matching $\boxed{\forall P.\forall R \vdash \mathcal{M} \succeq \mathcal{M}' \downarrow \varphi}$

$$\frac{\forall P.\forall R \vdash \mathcal{S} \succeq \mathcal{S}' \downarrow \varphi}{\forall P.\forall R \vdash \mathcal{S} \succeq \mathcal{S}' \downarrow \varphi} \quad (M\text{-}7) \qquad \frac{\forall P.\forall R \vdash \mathcal{F} \succeq \mathcal{F}' \downarrow \varphi}{\forall P.\forall R \vdash \mathcal{F} \succeq \mathcal{F}' \downarrow \varphi} \quad (M\text{-}8)$$

Figure 5.19: An algorithm for matching. Subject to certain constraints, we have $\forall P.\forall R \vdash \mathcal{O} \succeq \mathcal{O}' \downarrow \varphi$ if, and only if, $\mathcal{O} \succeq \varphi\,(\mathcal{O}')$.

will have:

$$\mathcal{O} \succeq \varphi\left(\mathcal{O}'\right) \quad \text{if, and only if,} \quad \forall P.\forall R \vdash \mathcal{O} \succeq \mathcal{O}' \downarrow \varphi.$$

The algorithm traverses the structure of $\mathcal{O}'$ to incrementally compute a matching realisation while verifying enrichment. Informally, Rule (M-2) verifies that $\mathcal{S}$ defines an equivalent type component $\mathbf{t}$, where $\mathbf{t}$ has a specific definition in $\mathcal{S}'$. Rule (M-3), on the other hand, verifies that $\mathcal{S}$ realises the type component $\mathbf{t}$ of $\mathcal{S}'$, where $\mathbf{t}$ is merely specified, but not defined, in $\mathcal{S}'$. More precisely, $\mathbf{t}$ is specified in way that permits it to have *some* functional dependency α on the fixed parameters $\beta_0, \ldots, \beta_{n-1}$. The type $\alpha\ \beta_0 \cdots \beta_{n-1}$ corresponds to a higher-order pattern in the sense of Miller [Mil91]: α is a variable that must occur in the domain of the computed realisation; its arguments $\{\beta_0, \ldots, \beta_{n-1}\} \subseteq R$ are fixed parameters on which the definition of $\mathbf{t}$ may depend. The preconditions on the inputs to the algorithm will ensure that the side condition $\alpha \notin P \cup R$ uniquely determines whether Rule (M-2) or Rule (M-3) applies. In Rule M-6, due to the contravariance of functor enrichment, the computation of the realisation φ is delayed as the roles of left and right object are swapped in order to construct a realisation φ' with domain N such that the generic domain of the lesser functor matches the realised domain of the richer functor.

The rules define an algorithm because, on any input satisfying the preconditions, there is a most one instance of a rule that applies (the rules are syntax directed). Further details of the algorithm's operation and the roles of the input sets P and R will be explained in Section 5.6.1.

Theorem 5.26 (Termination). *The algorithm of Definition 5.25 terminates.*

Proof. *We define a positive measure on the size of the inputs to the algorithm and then show that in each rule, the size of the inputs to each premise is strictly smaller than the size of the inputs to its conclusion. In particular,* *we define*

$$
\begin{aligned}
m(\epsilon_{\mathcal{S}}) &\stackrel{\text{def}}{=} 0 \\
m(d = \tau, \mathcal{S}) &\stackrel{\text{def}}{=} 1 + m(\mathcal{S}) \\
m(\mathrm{x} : v, \mathcal{S}) &\stackrel{\text{def}}{=} 1 + m(\mathcal{S}) \\
m(\mathrm{X} : \mathcal{M}, \mathcal{S}) &\stackrel{\text{def}}{=} 1 + m(\mathcal{M}) + m(\mathcal{S}) \\
\\
m(\forall P.\mathcal{M} \to \mathcal{M}') &\stackrel{\text{def}}{=} 1 + m(\mathcal{M}) + m(\mathcal{M}') \\
\\
m(\mathcal{M}) &\stackrel{\text{def}}{=} \begin{cases} 1 + m(\mathcal{S}) & \text{if } \mathcal{M} \equiv \mathcal{S} \\ 1 + m(\mathcal{F}) & \text{if } \mathcal{M} \equiv \mathcal{F} \end{cases} \\
m(P, R, \mathcal{O}, \mathcal{O}') &\stackrel{\text{def}}{=} m(\mathcal{O}) + m(\mathcal{O}').
\end{aligned}
$$

Observing that $m(\varphi(\mathcal{O})) = m(\mathcal{O})$, it is easy to see that each invocation of the algorithm on inputs P, R, $\mathcal{O}$, and $\mathcal{O}'$ of size $m(P, R, \mathcal{O}, \mathcal{O}')$ decreases the size of this measure in recursive calls.

5.6.1 Ground and Solvable Modules

We now define conditions on the inputs to the algorithm for which it is well-behaved. In Section 5.7 we will show that these conditions are always satisfied whenever we need to invoke the algorithm. Intuitively, we will say a matching problem $\forall P.\forall R \vdash \mathcal{O} \succeq \mathcal{O}' \downarrow \_$ is well-posed, if $\mathcal{O}$ is *ground* and $\mathcal{O}'$, the object to which the desired realisation will be applied, is *solvable*. More precisely, we define:

Definition 5.27 (Ground and Solvable Modules). The family of predicates
$\forall\_ \vdash \_$ **Gnd** and $\forall\_.\exists\_.\forall\_ \vdash \_$ **Slv** on structures, functors and modules is defined by the rules in Figure 5.20 and 5.21. The predicates characterise the classes of *ground* and *solvable* objects (respectively) for which Algorithm 5.25 is well-behaved.

Intuitively, if an object $\mathcal{O}$ is ground with respect to a set of variables P, written $\forall P \vdash \mathcal{O}$ **Gnd**, then $\mathrm{FV}(\mathcal{O}) \subseteq P$. Moreover, if $\mathcal{O}$ is a functor $\forall Q.\mathcal{M} \to \mathcal{M}'$ then, because of contravariance (and thus the swapping of roles in Rule *M*-6), we also require that $\mathcal{M}$ is solvable for the variables in Q, and (swapping back again) that $\mathcal{M}'$ is ground with respect to $P \cup Q$ (note we must cater for the possible occurrence of the parameters Q in $\mathcal{M}'$).

Ground Structures $\boxed{\forall P \vdash \mathcal{S}\ \mathbf{Gnd}}$

$$\frac{\begin{array}{l}\forall \mathrm{t} \in \mathrm{Dom}(\mathcal{S}).\ \mathrm{FV}(\mathcal{S}(\mathrm{t})) \subseteq P \\ \forall \mathrm{x} \in \mathrm{Dom}(\mathcal{S}).\ \mathrm{FV}(\mathcal{S}(\mathrm{x})) \subseteq P \\ \forall \mathrm{X} \in \mathrm{Dom}(\mathcal{S}).\ \forall P \vdash \mathcal{S}(\mathrm{X})\ \mathbf{Gnd}\end{array}}{\forall P \vdash \mathcal{S}\ \mathbf{Gnd}} \quad (G\text{-}1)$$

Ground Functors $\boxed{\forall P \vdash \mathcal{F}\ \mathbf{Gnd}}$

$$\frac{\forall P.\exists Q.\forall \emptyset \vdash \mathcal{M}_Q\ \mathbf{Slv} \quad \forall P \cup Q \vdash \mathcal{M}'_Q\ \mathbf{Gnd}}{\forall P \vdash \forall Q.\mathcal{M}_Q \rightarrow \mathcal{M}'_Q\ \mathbf{Gnd}} \quad (G\text{-}2)$$

Ground Modules $\boxed{\forall P \vdash \mathcal{M}\ \mathbf{Gnd}}$

$$\frac{\forall P \vdash \mathcal{S}\ \mathbf{Gnd}}{\forall P \vdash \mathcal{S}\ \mathbf{Gnd}} \quad (G\text{-}3)$$

$$\frac{\forall P \vdash \mathcal{F}\ \mathbf{Gnd}}{\forall P \vdash \mathcal{F}\ \mathbf{Gnd}} \quad (G\text{-}4)$$

Figure 5.20: The definition of ground and solvable objects (the definition of solvable objects is continued in Figure 5.21). We will show that provided $\forall P \cup R \vdash \mathcal{O}\ \mathbf{Gnd}$ and $\forall P.\exists Q.\forall R \vdash \mathcal{O}'\ \mathbf{Slv}$, then, for any realisation φ with $\mathrm{Dom}(\varphi) = Q$ and $\mathrm{Reg}(\varphi) \cap R = \emptyset$, we have $\forall P.\forall R \vdash \mathcal{O} \succeq \mathcal{O}' \ \downarrow\ \varphi$ if, and only if, $\mathcal{O} \succeq \varphi\,(\mathcal{O}')$.

Solvable Structures $\boxed{\forall P.\exists Q.\forall R \vdash \mathcal{S}\ \textbf{Slv}}$

$$\frac{P \cap R = \emptyset}{\forall P.\exists \emptyset.\forall R \vdash \epsilon_{\mathcal{S}}\ \textbf{Slv}} \tag{S-1}$$

$$\frac{\mathrm{FV}(\tau) \subseteq P \cup R \quad \forall P.\exists Q.\forall R \vdash \mathcal{S}\ \textbf{Slv}}{\forall P.\exists Q.\forall R \vdash \mathrm{t} = \tau, \mathcal{S}\ \textbf{Slv}} \tag{S-2}$$

$$\frac{\begin{array}{l}\alpha \notin P \\ \forall i \in [n].\beta_i \in R \\ \forall i \neq j \in [n].\beta_i \neq \beta_j \\ \forall P \cup \{\alpha\}.\exists Q.\forall R \vdash \mathcal{S}\ \textbf{Slv}\end{array}}{\forall P.\exists \{\alpha\} \cup Q.\forall R \vdash \mathrm{t} = \alpha\ \beta_0 \cdots \beta_{n-1}, \mathcal{S}\ \textbf{Slv}} \tag{S-3}$$

$$\frac{\mathrm{FV}(v) \subseteq P \cup R \quad \forall P.\exists Q.\forall R \vdash \mathcal{S}\ \textbf{Slv}}{\forall P.\exists Q.\forall R \vdash \mathrm{x} : v, \mathcal{S}\ \textbf{Slv}} \tag{S-4}$$

$$\frac{\forall P.\exists Q.\forall R \vdash \mathcal{M}\ \textbf{Slv} \quad \forall P \cup Q.\exists Q'.\forall R \vdash \mathcal{S}\ \textbf{Slv}}{\forall P.\exists Q \cup Q'.\forall R \vdash \mathrm{X} : \mathcal{M}, \mathcal{S}\ \textbf{Slv}} \tag{S-5}$$

Solvable Functors $\boxed{\forall P.\exists Q.\forall R \vdash \mathcal{F}\ \textbf{Slv}}$

$$\frac{\forall P \cup R.\exists N.\forall \emptyset \vdash M\ \textbf{Slv} \quad \forall P.\exists Q.\forall R \cup N \vdash M'\ \textbf{Slv}}{\forall P.\exists Q.\forall R \vdash \forall N.M \to M'\ \textbf{Slv}} \tag{S-6}$$

Solvable Modules $\boxed{\forall P.\exists Q.\forall R \vdash \mathcal{M}\ \textbf{Slv}}$

$$\frac{\forall P.\exists Q.\forall R \vdash \mathcal{S}\ \textbf{Slv}}{\forall P.\exists Q.\forall R \vdash \mathcal{S}\ \textbf{Slv}} \tag{S-7}$$

$$\frac{\forall P.\exists Q.\forall R \vdash \mathcal{F}\ \textbf{Slv}}{\forall P.\exists Q.\forall R \vdash \mathcal{F}\ \textbf{Slv}} \tag{S-8}$$

Figure 5.21: The definition of solvable objects (continued from Figure 5.20).

Solvability is captured by the predicate $\forall P.\exists Q.\forall R \vdash \mathcal{O}$ **Slv**. Suppose we are looking for a realisation φ as a *solution* to a matching problem. Intuitively, the prefix $\forall P.\exists Q.\forall R$ declares the role of any free type variables in $\mathcal{O}$. The set P contains variables which *may* occur free in $\mathrm{Reg}(\varphi)$. Q is the set of variables we are solving for, i.e. we require $\mathrm{Dom}(\varphi) = Q$. The set R lists the parameters which *must not* occur free in $\mathrm{Reg}(\varphi)$, i.e. it records the set of parameters introduced by any enclosing semantic functor of $\mathcal{O}$, and thus the parameters that may appear as arguments to variables in Q. However, if $\mathcal{O}$ is a functor $\forall N.\mathcal{M} \to \mathcal{M}'$ then, because of contravariance (and thus the swapping of roles in Rule M-6), we require that $\mathcal{M}$ is solvable for the variables in N (with respect to free variables $P \cup R$ and no parameters), and, swapping back again, $\mathcal{M}'$ is solvable for Q with respect to P, and the parameters $R \cup N$. The set N is added to R as we enter the range of the functor: since we are now in the scope of N, variables in Q should be allowed to take on these additional parameters.

We can characterise the solvability of $\mathcal{O}$ for a set a variables Q, with respect to free variables P and parameters R, informally as follows:

- The sets P, Q, and R are distinct.
- $\mathrm{FV}(\mathcal{O}) \subseteq P \cup Q \cup R$.
- $\mathrm{FV}(\mathcal{O}) \setminus (P \cup R) = Q$.
- If $\mathcal{O}$ is a functor $\forall N.\mathcal{M} \to \mathcal{M}'$ then its domain $\mathcal{M}$ must be solvable for the variables N, with respect to free variables $P \cup R$ and no parameters. On the one hand, the parameters R of Q are considered as free variables for N, because the variables in N are declared within the scope of R. On the other hand, because no variables have yet been declared in the scope of N, the set of parameters for N is empty.
- Each variable $\alpha \in Q$ *first occurs positively* within $\mathcal{O}$, where α *first occurs positively* in $\mathcal{O}$ if, and only if:

  either $\mathcal{O}$ is a structure $\mathcal{S}$, and

  either α first occurs in a type binding $\mathrm{t} = \alpha\ \beta_0 \cdots \beta_{n-1}$ within $\mathcal{S}$, where $\beta_0, \ldots, \beta_{n-1}$ are distinct parameters drawn from R.

  or α first occurs positively within a sub-module of $\mathcal{S}$;

  or $\mathcal{O}$ is a functor $\forall N.\mathcal{M} \to \mathcal{M}'$, in which case α may not occur in the domain $\mathcal{M}$, and must first occur positively in the range $\mathcal{M}'$, where it may be applied to any of the additional parameters in N as well as R.

Our algorithm $\forall P.\forall R \vdash \mathcal{O} \succeq \mathcal{O}' \downarrow \_$ operates by traversing $\mathcal{O}'$ to find a first positive occurrence of a variable not occurring in $P \cup R$, matching it against its corresponding binding in $\mathcal{O}$ to determine its realisation, applying the realisation to the remaining problem, and proceeding until the traversal of $\mathcal{O}'$ and the construction of the realisation is complete. Along the way, it also checks that the enrichment relation holds. The two conditions, that the region of the realisation may not contain parameters (from R) and that each variable in the domain of the realisation is applied to a list of distinct parameters, ensure that the realisation is unique, provided it exists.

Properties of Ground and Solvable Objects

It is easy to verify the following properties of ground and solvable objects. We shall make use of them in subsequent proofs.

Lemma 5.28 (Closure).

- $\forall P \vdash \mathcal{O}\ \mathbf{Gnd} \supset \mathrm{FV}(\mathcal{O}) \subseteq P$.
- $\forall P.\exists Q.\forall R \vdash \mathcal{O}\ \mathbf{Slv} \supset P \cap Q = \emptyset \wedge Q \cap R = \emptyset \wedge P \cap R = \emptyset$.
- $\forall P.\exists Q.\forall R \vdash \mathcal{O}\ \mathbf{Slv} \supset \mathrm{FV}(\mathcal{O}) \subseteq P \cup Q \cup R$.
- $\forall P.\exists Q.\forall R \vdash \mathcal{O}\ \mathbf{Slv} \supset \mathrm{FV}(\mathcal{O}) \setminus (P \cup R) = Q$.

Lemma 5.29 (Strengthening).

- *If* $\forall P \cup \mathrm{Dom}(\varphi) \vdash \mathcal{O}\ \mathbf{Gnd}$, $\mathrm{Dom}(\varphi) \cap P = \emptyset$ *and* $\mathrm{Reg}(\varphi) \subseteq P$ *then* $\forall P \vdash \varphi\,(\mathcal{O})\ \mathbf{Gnd}$.
- *If* $\forall P \cup \mathrm{Dom}(\varphi).\exists Q.\forall R \vdash \mathcal{O}\ \mathbf{Slv}$, $\mathrm{Dom}(\varphi) \cap P = \emptyset$ *and* $\mathrm{Reg}(\varphi) \subseteq P \cup R$ *then* $\forall P.\exists Q.\forall R \vdash \varphi\,(\mathcal{O})\ \mathbf{Slv}$.
- *In particular, if* $\forall P \cup \{\alpha^\kappa\}.\exists Q.\forall R \vdash \mathcal{O}\ \mathbf{Slv}$, $\alpha^\kappa \notin P$ *and* $\mathrm{FV}(\tau^\kappa) \subseteq P \cup R$ *then* $\forall P.\exists Q.\forall R \vdash [\tau^\kappa/\alpha^\kappa]\,(\mathcal{O})\ \mathbf{Slv}$.

Lemma 5.30 (Weakening).

- $\forall P \vdash \mathcal{O}\ \mathbf{Gnd} \supset \forall P \cup P' \vdash \mathcal{O}\ \mathbf{Gnd}$.
- $\forall P.\exists Q.\forall R \vdash \mathcal{O}\ \mathbf{Slv} \supset P' \cap (Q \cup R) = \emptyset \supset \forall P \cup P'.\exists Q.\forall R \vdash \mathcal{O}\ \mathbf{Slv}$.

Lemma 5.31 (Grounding).

- $\forall P.\exists Q.\forall R \vdash \mathcal{O}\ \mathbf{Slv} \supset \forall P \cup Q \cup R \vdash \mathcal{O}\ \mathbf{Gnd}$.

5.6.2 Soundness

We can now verify that our algorithm only computes correct realisations:

Theorem 5.32 (Soundness). *If* $\forall P.\forall R \vdash \mathcal{O} \succeq \mathcal{O}' \downarrow \varphi$ *then* $\mathcal{O} \succeq \varphi(\mathcal{O}')$, *provided* $\forall P \cup R \vdash \mathcal{O}$ **Gnd** *and* $\forall P.\exists \mathrm{Dom}(\varphi).\forall R \vdash \mathcal{O}'$ **Slv**.

(Readers not interested in the proof should skip ahead to Section 5.6.3 on page 205.)

Proof (Soundness). *We prove the stronger theorem:*

$$\begin{array}{l}\forall P.\forall R \vdash \mathcal{O} \succeq \mathcal{O}' \downarrow \varphi \supset \\ \quad \forall P \cup R \vdash \mathcal{O}\ \mathbf{Gnd} \supset \\ \quad\quad \forall Q.\ \forall P.\exists Q.\forall R \vdash \mathcal{O}'\ \mathbf{Slv} \supset \\ \quad\quad \left(\begin{array}{l}\mathrm{Dom}(\varphi) = Q \\ \wedge \mathrm{Reg}(\varphi) \subseteq P \\ \wedge \mathcal{O} \succeq \varphi(\mathcal{O}')\end{array}\right)\end{array}.$$

by induction on the rules in Figure 5.19.

$\boxed{M\text{-}\mathbf{1}}$ *Easy.*

$\boxed{M\text{-}\mathbf{2}}$ *By induction we may assume:*

$$\mathrm{t} \in \mathrm{Dom}(\mathcal{S}), \tag{1}$$

$$\mathcal{S}(\mathrm{t}) = \tau, \tag{2}$$

$$\begin{array}{l}\forall P \cup R \vdash \mathcal{S}\ \mathbf{Gnd} \supset \\ \quad \forall Q.\ \forall P.\exists Q.\forall R \vdash \mathcal{S}'\ \mathbf{Slv} \supset \\ \quad \left(\begin{array}{l}\mathrm{Dom}(\varphi) = Q \\ \wedge \mathrm{Reg}(\varphi) \subseteq P \\ \wedge \mathcal{S} \succeq \varphi(\mathcal{S}')\end{array}\right)\end{array} \tag{3}$$

We need to show:

$$\begin{array}{l}\forall P \cup R \vdash \mathcal{S}\ \mathbf{Gnd} \supset \\ \quad \forall Q.\ \forall P.\exists Q.\forall R \vdash \mathrm{t} = \tau, \mathcal{S}'\ \mathbf{Slv} \supset \\ \quad \left(\begin{array}{l}\mathrm{Dom}(\varphi) = Q \\ \wedge \mathrm{Reg}(\varphi) \subseteq P \\ \wedge \mathcal{S} \succeq \varphi(\mathrm{t} = \tau, \mathcal{S}')\end{array}\right)\end{array}$$

Assume

$$\forall P \cup R \vdash \mathcal{S}\ \mathbf{Gnd}. \tag{4}$$

Consider an arbitrary Q such that:

$$\forall P.\exists Q.\forall R \vdash \mathrm{t} = \tau, \mathcal{S}'\ \mathbf{Slv}. \tag{5}$$

Inverting (4)*, by* (1) *we have, in particular* $\mathrm{FV}(\mathcal{S}(\mathrm{t})) \subseteq P \cup R$*. Hence, by* (2)*:*

$$\mathrm{FV}(\tau) \subseteq P \cup R. \tag{6}$$

Lemma 5.28 (Closure) on (5) *ensures* $Q \cap (P \cup R) = \emptyset$*, from which*

$$Q \cap (\mathrm{FV}(\tau)) = \emptyset \tag{7}$$

follows by (6)*. Hence* (5) *cannot have been derived by Rule* ($\mathcal{S}$-3)*, and must in fact result from an application of Rule* ($\mathcal{S}$-2)*. Inverting* (5) *we therefore have both*

$$\mathrm{FV}(\tau) \subseteq P \cup R \tag{8}$$

and

$$\forall P.\exists Q.\forall R \vdash \mathcal{S}'\ \mathbf{Slv}. \tag{9}$$

We can now apply induction hypothesis (3) *to* (4)*, Q, and* (9) *to obtain:*

$$\mathrm{Dom}(\varphi) = Q, \tag{10}$$

$$\mathrm{Reg}(\varphi) \subseteq P \tag{11}$$

and

$$\mathcal{S} \succeq \varphi\left(\mathcal{S}'\right). \tag{12}$$

Clearly, by (10) *we can re-express* (7) *as* $\mathrm{Dom}(\varphi) \cap \mathrm{FV}(\tau) = \emptyset$*. Consequently:*

$$\varphi\left(\tau\right) = \tau. \tag{13}$$

With (2) *and* (12)*, we can verify the premises of Rule* ($\succeq$-1) *to derive:*

$$\mathcal{S} \succeq \mathrm{t} = \tau, \varphi\left(\mathcal{S}'\right), \tag{14}$$

which we may re-express by (13) *as*

$$\mathcal{S} \succeq \varphi\left(\mathrm{t} = \tau, \mathcal{S}'\right). \tag{15}$$

Combining (10)*,* (11) *and* (15) *gives the desired result.*

$\boxed{M\text{-}\mathbf{3}}$ *By induction we may assume:*

$$\alpha \notin P \cup R, \tag{1}$$

$$\mathrm{t} \in \mathrm{Dom}(\mathcal{S}), \tag{2}$$

$$\mathcal{S}(\mathrm{t}), (\alpha\ \beta_0 \cdots \beta_{n-1}) \in \mathit{Typ}^{\kappa}, \tag{3}$$

$$\mathrm{FV}(\mathcal{S}(\mathrm{t})) \cap R \subseteq \{\beta_i \mid i \in [n]\}, \tag{4}$$

$$\begin{array}{l}\forall P \cup R \vdash \mathcal{S}\ \mathbf{Gnd} \supset \\ \quad \forall Q.\ \ \forall P.\exists Q.\forall R \vdash [\Lambda\beta_0 \cdots \beta_{n-1}.\mathcal{S}(\mathrm{t})/\alpha]\,(\mathcal{S}')\ \mathbf{Slv} \supset \\ \quad\ \left(\begin{array}{l}\mathrm{Dom}(\varphi) = Q \\ \wedge\, \mathrm{Reg}(\varphi) \subseteq P \\ \wedge\, \mathcal{S} \succeq \varphi\left([\Lambda\beta_0 \cdots \beta_{n-1}.\mathcal{S}(\mathrm{t})/\alpha]\,(\mathcal{S}')\right)\end{array}\right)\end{array} \tag{5}$$

We need to show:

$$\begin{array}{l}\forall P \cup R \vdash \mathcal{S}\ \mathbf{Gnd} \supset \\ \quad \forall Q.\ \ \forall P.\exists Q.\forall R \vdash \mathrm{t} = \alpha\ \beta_0 \cdots \beta_{n-1}, \mathcal{S}'\ \mathbf{Slv} \supset \\ \quad\ \left(\begin{array}{l}\mathrm{Dom}\left([\Lambda\beta_0 \cdots \beta_{n-1}.\mathcal{S}(\mathrm{t})/\alpha] \mid \varphi\right) = Q \\ \wedge\, \mathrm{Reg}\left([\Lambda\beta_0 \cdots \beta_{n-1}.\mathcal{S}(\mathrm{t})/\alpha] \mid \varphi\right) \subseteq P \\ \wedge\, \mathcal{S} \succeq \left([\Lambda\beta_0 \cdots \beta_{n-1}.\mathcal{S}(\mathrm{t})/\alpha] \mid \varphi\right)\left(\mathrm{t} = \alpha\ \beta_0 \cdots \beta_{n-1}, \mathcal{S}'\right)\end{array}\right)\end{array}$$

Assume

$$\forall P \cup R \vdash \mathcal{S}\ \mathbf{Gnd}. \tag{6}$$

Consider an arbitrary Q such that

$$\forall P.\exists Q.\forall R \vdash \mathrm{t} = \alpha\ \beta_0 \cdots \beta_{n-1}, \mathcal{S}'\ \mathbf{Slv}. \tag{7}$$

By (1) *we know that* $\mathrm{FV}(\alpha\ \beta_0 \cdots \beta_{n-1}) \not\subseteq P \cup R$ *hence* (7) *cannot have been derived by Rule* (*S*-2) *and must be the result of an application of Rule* (*S*-3)*. Inverting* (7) *we therefore have:*

$$\alpha \notin P, \tag{8}$$

$$\forall i \in [n].\beta_i \in R, \tag{9}$$

$$\forall i \neq j \in [n].\beta_i \neq \beta_j, \tag{10}$$

$$\forall P \cup \{\alpha\}.\exists Q'.\forall R \vdash \mathcal{S}' \ \mathbf{Slv}, \tag{11}$$

for some Q' *with* $Q = \{\alpha\} \cup Q'$.

Inverting (6), *by* (2) *we have* $\mathrm{FV}(\mathcal{S}(\mathrm{t})) \subseteq P \cup R$, *which together with* (4) *ensures:*

$$\mathrm{FV}(\Lambda\beta_0 \cdots \beta_{n-1}.\mathcal{S}(\mathrm{t})) \subseteq P. \tag{12}$$

Lemma 5.29 *(Strengthening) on* (11) *with* $[\Lambda\beta_0 \cdots \beta_{n-1}.\mathcal{S}(\mathrm{t})/\alpha]$ *using* (8) *and* (12) *produces:*

$$\forall P.\exists Q'.\forall R \vdash [\Lambda\beta_0 \cdots \beta_{n-1}.\mathcal{S}(\mathrm{t})/\alpha]\left(\mathcal{S}'\right) \mathbf{Slv}. \tag{13}$$

We can now apply induction hypothesis (5) *to* (6), Q' *and* (13) *to obtain:*

$$\mathrm{Dom}(\varphi) = Q', \tag{14}$$

$$\mathrm{Reg}(\varphi) \subseteq P \tag{15}$$

and

$$\mathcal{S} \succeq \varphi\left([\Lambda\beta_0 \cdots \beta_{n-1}.\mathcal{S}(\mathrm{t})/\alpha]\left(\mathcal{S}'\right)\right). \tag{16}$$

It is straightforward to show:

$$\mathrm{Dom}([\Lambda\beta_0 \cdots \beta_{n-1}.\mathcal{S}(\mathrm{t})/\alpha] \mid \varphi) = \{\alpha\} \cup \mathrm{Dom}(\varphi) = Q. \tag{17}$$

Moreover, (12) *and* (15) *ensure that*

$$\mathrm{Reg}([\Lambda\beta_0 \cdots \beta_{n-1}.\mathcal{S}(\mathrm{t})/\alpha] \mid \varphi) \subseteq P. \tag{18}$$

Lemma 5.28 *(Closure) on* (11) *yields:*

$$Q' \cap ((P \cup \{\alpha\}) \cup R) = \emptyset. \tag{19}$$

Together with (19), (14) *and* (9) *ensure that*

$$\alpha \notin \mathrm{Dom}(\varphi), \tag{20}$$

$$\forall i \in [n].\beta_i \notin \mathrm{Dom}([\Lambda\beta_0 \cdots \beta_{n-1}.\mathcal{S}(\mathrm{t})/\alpha] \mid \varphi). \tag{21}$$

By applying the realisation using facts (20) *and* (21) *we can verify:*

$$\mathcal{S}(\mathrm{t}) = ([\Lambda\beta_0 \cdots \beta_{n-1}.\mathcal{S}(\mathrm{t})/\alpha] \mid \varphi)\left(\alpha\ \beta_0 \cdots \beta_{n-1}\right). \tag{22}$$

Moreover, from (19), (12) *and* (14) *we can verify that* $\mathrm{Dom}(\varphi) \cap \mathrm{Inv}([\Lambda\beta_0 \cdots \beta_{n-1}.\mathcal{S}(\mathrm{t})/\alpha]) = \emptyset$*. Hence*

$$([\Lambda\beta_0 \cdots \beta_{n-1}.\mathcal{S}(\mathrm{t})/\alpha] \mid \varphi)\left(\mathcal{S}'\right) = \varphi\left([\Lambda\beta_0 \cdots \beta_{n-1}.\mathcal{S}(\mathrm{t})/\alpha]\left(\mathcal{S}'\right)\right),$$

and thus, by equation (22)*:*

$$\begin{aligned}&\left(\mathrm{t} = \mathcal{S}(\mathrm{t}), \varphi\left([\Lambda\beta_0 \cdots \beta_{n-1}.\mathcal{S}(\mathrm{t})/\alpha]\left(\mathcal{S}'\right)\right)\right)\\ &\qquad = ([\Lambda\beta_0 \cdots \beta_{n-1}.\mathcal{S}(\mathrm{t})/\alpha] \mid \varphi)\left(\mathrm{t} = \alpha\ \beta_0 \cdots \beta_{n-1}, \mathcal{S}'\right).\end{aligned} \tag{23}$$

With (2), (22) *and* (16)*, we can verify the premises of Rule* ($\succeq$-1) *that, if followed by equation* (23)*, derives:*

$$\mathcal{S} \succeq ([\Lambda\beta_0 \cdots \beta_{n-1}.\mathcal{S}(\mathrm{t})/\alpha] \mid \varphi)\left(\mathrm{t} = \alpha\ \beta_0 \cdots \beta_{n-1}, \mathcal{S}'\right). \tag{24}$$

Combining (17), (18) *and* (24) *gives the desired result.*

$\boxed{M\textbf{-4}}$ *Similar to, but easier than, case* $\boxed{M\textbf{-2}}$.

$\boxed{M\textbf{-5}}$ *By induction we may assume:*

$$\mathrm{X} \in \mathrm{Dom}(\mathcal{S}), \tag{1}$$

$$\begin{aligned}&\forall P \cup R \vdash \mathcal{S}(\mathrm{X})\ \mathbf{Gnd} \supset\\ &\quad \forall Q.\ \forall P.\exists Q.\forall R \vdash \mathcal{M}\ \mathbf{Slv} \supset\\ &\quad \begin{pmatrix} \mathrm{Dom}(\varphi_1) = Q\\ \wedge\, \mathrm{Reg}(\varphi_1) \subseteq P\\ \wedge\, \mathcal{S}(\mathrm{X}) \succeq \varphi_1(\mathcal{M}) \end{pmatrix}\end{aligned} \tag{2}$$

$$\begin{aligned}&\forall P \cup R \vdash \mathcal{S}\ \mathbf{Gnd} \supset\\ &\quad \forall Q.\ \forall P.\exists Q.\forall R \vdash \varphi_1(\mathcal{S}')\ \mathbf{Slv} \supset\\ &\quad \begin{pmatrix} \mathrm{Dom}(\varphi_2) = Q\\ \wedge\, \mathrm{Reg}(\varphi_2) \subseteq P\\ \wedge\, \mathcal{S} \succeq \varphi_2(\varphi_1(\mathcal{S}')) \end{pmatrix}\end{aligned} \tag{3}$$

We need to show:

$$\begin{aligned}&\forall P \cup R \vdash \mathcal{S}\ \mathbf{Gnd} \supset\\ &\quad \forall Q.\ \forall P.\exists Q.\forall R \vdash \mathrm{X} : \mathcal{M}, \mathcal{S}'\ \mathbf{Slv} \supset\\ &\quad \begin{pmatrix} \mathrm{Dom}(\varphi_1 \mid \varphi_2) = Q\\ \wedge\, \mathrm{Reg}(\varphi_1 \mid \varphi_2) \subseteq P\\ \wedge\, \mathcal{S} \succeq (\varphi_1 \mid \varphi_2)(\mathrm{X} : \mathcal{M}, \mathcal{S}') \end{pmatrix}\end{aligned}$$

Assume
$$\forall P \cup R \vdash \mathcal{S}\ \mathbf{Gnd}. \tag{4}$$
Consider an arbitrary Q such that
$$\forall P.\exists Q.\forall R \vdash \mathrm{X} : \mathcal{M}, \mathcal{S}'\ \mathbf{Slv}. \tag{5}$$
Inverting (5) we must have both
$$\forall P.\exists Q_1.\forall R \vdash \mathcal{M}\ \mathbf{Slv} \tag{6}$$
and
$$\forall P \cup Q_1.\exists Q_2.\forall R \vdash \mathcal{S}'\ \mathbf{Slv}. \tag{7}$$
for some Q_1, Q_2 with $Q = Q_1 \cup Q_2$.
Inverting (4) together with (1) yields
$$\forall P \cup R \vdash \mathcal{S}(\mathrm{X})\ \mathbf{Gnd}. \tag{8}$$
Applying induction hypothesis (2) to (8), Q_1, and (6) we obtain:
$$\mathrm{Dom}(\varphi_1) = Q_1, \tag{9}$$
$$\mathrm{Reg}(\varphi_1) \subseteq P, \tag{10}$$
$$\mathcal{S}(\mathrm{X}) \succeq \varphi_1(\mathcal{M}). \tag{11}$$
Lemma 5.28 (Closure) on (6) ensures
$$Q_1 \cap P = \emptyset \tag{12}$$
and
$$\mathrm{FV}(\mathcal{M}) \subseteq P \cup Q_1 \cup R. \tag{13}$$
Lemma 5.29 (Strengthening) on (7) with φ_1 using (9), (12) and (10) yields:
$$\forall P.\exists Q_2.\forall R \vdash \varphi_1\left(\mathcal{S}'\right)\ \mathbf{Slv}. \tag{14}$$
We can now apply induction hypothesis (3) to (4), Q_2, and (14) to obtain:
$$\mathrm{Dom}(\varphi_2) = Q_2, \tag{15}$$

$$\mathrm{Reg}(\varphi_2) \subseteq P, \tag{16}$$

$$\mathcal{S} \succeq \varphi_2\left(\varphi_1\left(\mathcal{S}'\right)\right). \tag{17}$$

Lemma 5.28 (Closure) on (7) ensures

$$Q_2 \cap (P \cup Q_1 \cup R) = \emptyset. \tag{18}$$

It is easy to show

$$\mathrm{Dom}(\varphi_1 \mid \varphi_2) = Q, \tag{19}$$

$$\mathrm{Reg}(\varphi_1 \mid \varphi_2) \subseteq P. \tag{20}$$

Now $(\varphi_1 \mid \varphi_2)(\mathcal{M}) = \varphi_1(\mathcal{M})$ *follows from (13), (15) and (18). Moreover, we can easily verify that* $\mathrm{Dom}(\varphi_2) \cap \mathrm{Inv}(\varphi_1) = \emptyset$ *using (9), (10) and (18). Hence* $(\varphi_1 \mid \varphi_2)(\mathcal{S}') = \varphi_2(\varphi_1(\mathcal{S}'))$ *and we have:*

$$(\varphi_1 \mid \varphi_2)\left(\mathrm{X} : \mathcal{M}, \mathcal{S}'\right) = \mathrm{X} : \varphi_1(\mathcal{M}), \varphi_2\left(\varphi_1\left(\mathcal{S}'\right)\right). \tag{21}$$

With (1), (11), and (17) we can verify the premises of Rule ($\succeq$ *-1) that, if followed by equation (21), derives:*

$$\mathcal{S} \succeq (\varphi_1 \mid \varphi_2)\left(\mathrm{X} : \mathcal{M}, \mathcal{S}'\right). \tag{22}$$

Combining (19), (20) and (22) gives the desired result.

$\boxed{M\textbf{-6}}$ *By induction we may assume:*

$$N \cap (P \cup R \cup M) = \emptyset, \tag{1}$$

$$M \cap (P \cup R) = \emptyset, \tag{2}$$

$$\begin{array}{l} \forall P \cup R \cup M \cup \emptyset \vdash \mathcal{M}_M \ \mathbf{Gnd} \supset \\ \quad \forall Q.\ \ \forall P \cup R \cup M . \exists Q . \forall \emptyset \vdash \mathcal{M}_N \ \mathbf{Slv} \supset \\ \quad \left(\begin{array}{l} \mathrm{Dom}(\varphi') = Q \\ \wedge \mathrm{Reg}(\varphi') \subseteq P \cup R \cup M \\ \wedge \mathcal{M}_M \succeq \varphi'(\mathcal{M}_N) \end{array} \right) \end{array} \tag{3}$$

$$\begin{array}{l} \forall P \cup R \cup M \vdash \varphi'(\mathcal{M}'_N)\ \mathbf{Gnd} \supset \\ \quad \forall Q.\ \ \forall P.\exists Q.\forall R \cup M \vdash \mathcal{M}'_M\ \mathbf{Slv} \supset \\ \quad\quad \left(\begin{array}{l} \mathrm{Dom}(\varphi) = Q \\ \wedge \mathrm{Reg}(\varphi) \subseteq P \\ \wedge \varphi'(\mathcal{M}'_N) \succeq \varphi(\mathcal{M}'_M) \end{array} \right) \end{array} \tag{4}$$

We need to show:

$$\begin{array}{l} \forall P \cup R \vdash \forall N.\mathcal{M}_N \to \mathcal{M}'_N\ \mathbf{Gnd} \supset \\ \quad \forall Q.\ \ \forall P.\exists Q.\forall R \vdash \forall M.\mathcal{M}_M \to \mathcal{M}'_M\ \mathbf{Slv} \supset \\ \quad\quad \left(\begin{array}{l} \mathrm{Dom}(\varphi) = Q \\ \wedge \mathrm{Reg}(\varphi) \subseteq P \\ \wedge \forall N.\mathcal{M}_N \to \mathcal{M}'_N \succeq \varphi(\forall M.\mathcal{M}_M \to \mathcal{M}'_M) \end{array} \right) \end{array}$$

Assume

$$\forall P \cup R \vdash \forall N.\mathcal{M}_N \to \mathcal{M}'_N\ \mathbf{Gnd}. \tag{5}$$

Consider an arbitrary Q such that

$$\forall P.\exists Q.\forall R \vdash \forall M.\mathcal{M}_M \to \mathcal{M}'_M\ \mathbf{Slv}. \tag{6}$$

Inverting (6) we must have both

$$\forall P \cup R.\exists M.\forall \emptyset \vdash \mathcal{M}_M\ \mathbf{Slv} \tag{7}$$

and

$$\forall P.\exists Q.\forall R \cup M \vdash \mathcal{M}'_M\ \mathbf{Slv}. \tag{8}$$

Inverting (5) we must have both

$$\forall P \cup R.\exists N.\forall \emptyset \vdash \mathcal{M}_N\ \mathbf{Slv} \tag{9}$$

and

$$\forall P \cup R \cup N \vdash \mathcal{M}'_N\ \mathbf{Gnd}. \tag{10}$$

Applying Lemma 5.31 (Grounding) to (7) we obtain:

$$\forall P \cup R \cup M \vdash \mathcal{M}_M\ \mathbf{Gnd}. \tag{11}$$

From (1), in particular, we have $M \cap N = \emptyset$, so by Lemma 5.30 (Weakening) on (9) and M we also have:

$$\forall P \cup R \cup M.\exists N.\forall \emptyset \vdash \mathcal{M}_N\ \mathbf{Slv}. \tag{12}$$

Induction hypothesis (3) on (11), N, and (12) produces:

$$\mathrm{Dom}(\varphi') = N, \tag{13}$$

$$\mathrm{Reg}(\varphi') \subseteq P \cup R \cup M, \tag{14}$$

$$\mathcal{M}_M \succeq \varphi'(\mathcal{M}_N). \tag{15}$$

By Lemma 5.30 (Weakening) on (10) and M we also have:

$$\forall P \cup R \cup N \cup M \vdash \mathcal{M}'_N \ \mathbf{Gnd}. \tag{16}$$

Furthermore, Lemma 5.29 (Strengthening) on (16) and φ' using (1), (13) and (14) establishes:

$$\forall P \cup R \cup M \vdash \varphi'\left(\mathcal{M}'_N\right) \ \mathbf{Gnd}. \tag{17}$$

We can now apply induction hypothesis (4) to (17), Q, and (8) to obtain:

$$\mathrm{Dom}(\varphi) = Q, \tag{18}$$

$$\mathrm{Reg}(\varphi) \subseteq P, \tag{19}$$

$$\varphi'\left(\mathcal{M}'_N\right) \succeq \varphi\left(\mathcal{M}'_M\right). \tag{20}$$

It remains to show:

$$\forall N.\mathcal{M}_N \to \mathcal{M}'_N \succeq \varphi\left(\forall M.\mathcal{M}_M \to \mathcal{M}'_M\right).$$

Two applications of Lemma 5.28 (Closure) on (7) and (8) ensure that $M \cap (P \cup R) = \emptyset$ and $M \cap Q = \emptyset$. Then, from (19) and (18), it is easy to see that $M \cap \mathrm{Inv}(\varphi) = \emptyset$. Hence:

$$\varphi\left(\forall M.\mathcal{M}_M \to \mathcal{M}'_M\right) = \forall M.\varphi\left(\mathcal{M}_M\right) \to \varphi\left(\mathcal{M}'_M\right). \tag{21}$$

Two applications of Lemma 5.28 (Closure) on (7) and (8) establish that $\mathrm{FV}(\mathcal{M}_M) \subseteq P \cup R \cup M$ and $Q \cap (P \cup R \cup M) = \emptyset$. Hence $\mathrm{Dom}(\varphi) \cap \mathrm{FV}(\mathcal{M}_M) = \emptyset$ and $\varphi\left(\mathcal{M}_M\right) = \mathcal{M}_M$ letting us re-express (15) as:

$$\varphi\left(\mathcal{M}_M\right) \succeq \varphi'\left(\mathcal{M}_N\right). \tag{22}$$

Two applications of Lemma 5.28 (Closure) on (5) and (7) allows us to verify that

$$M \cap \mathrm{FV}(\forall N.\mathcal{M}_N \to \mathcal{M}'_N) = \emptyset. \tag{23}$$

Applying Rule ($\succeq$ -2) to (22), (20), (13) and (23) derives:

$$\forall N.\mathcal{M}_N \to \mathcal{M}'_N \succeq \forall M.\varphi\left(\mathcal{M}_M\right) \to \varphi\left(\mathcal{M}'_M\right).$$

which we may re-express by (21) as:

$$\forall N.\mathcal{M}_N \to \mathcal{M}'_N \succeq \varphi\left(\forall M.\mathcal{M}_M \to \mathcal{M}'_M\right). \tag{24}$$

Combining (18), (19) and (24) gives the desired result.

$\boxed{M\text{-}\mathbf{7}}$ *Trivial induction.*

$\boxed{M\text{-}\mathbf{8}}$ *Trivial induction.*

5.6.3 Completeness

We can also show that our algorithm is complete, i.e. if a matching problem is well-posed and has a solution, then the algorithm computes it:

Theorem 5.33 (Completeness). *If $\mathcal{O} \succeq \varphi\left(\mathcal{O}'\right)$ then, provided $\forall P \cup R \vdash \mathcal{O}$ **Gnd** and $\forall P.\exists \mathrm{Dom}(\varphi).\forall R \vdash \mathcal{O}'$ **Slv** and $\mathrm{Reg}(\varphi) \cap R = \emptyset$, our algorithm succeeds with $\forall P.\forall R \vdash \mathcal{O} \succeq \mathcal{O}' \downarrow \varphi$.*

(Readers not interested in the proof of completeness should skip ahead to Section 5.7 on page 218.)

In order to prove the completeness of the matching algorithm we will need a stronger induction principle than the one provided by the definition of $\_ \succeq \_$. Fortunately, a slightly different definition of $\_ \succeq \_$ does the trick:

Definition 5.34 (Strong Enrichment). The family of relations $\_ \succeq' \_$ on structures, functors and modules is defined by the rules in Figure 5.22.

The alternative family of relations $\_ \succeq' \_$ gives us an appropriate induction principle for proving completeness. The validity of performing induction on the rules of $\_ \succeq' \_$ instead of $\_ \succeq \_$ is justified by:

Lemma 5.35 (Strong Induction).

$$\mathcal{O} \succeq \mathcal{O}' \supset \mathcal{O} \succeq' \mathcal{O}'$$

Structure Enrichment $\boxed{\mathcal{S} \succeq' \mathcal{S}'}$

$$\frac{}{\mathcal{S} \succeq' \epsilon_{\mathcal{S}}} \qquad (\succeq' \text{-1})$$

$$\frac{\mathrm{t} \in \mathrm{Dom}(\mathcal{S}) \quad \mathcal{S}(\mathrm{t}) = \tau \quad \mathcal{S} \succeq' \mathcal{S}'}{\mathcal{S} \succeq' \mathrm{t} = \tau, \mathcal{S}'} \qquad (\succeq' \text{-2})$$

$$\frac{\mathrm{x} \in \mathrm{Dom}(\mathcal{S}) \quad \mathcal{S}(\mathrm{x}) \succeq v \quad \mathcal{S} \succeq' \mathcal{S}'}{\mathcal{S} \succeq' \mathrm{x} : v, \mathcal{S}'} \qquad (\succeq' \text{-3})$$

$$\frac{\mathrm{X} \in \mathrm{Dom}(\mathcal{S}) \quad \mathcal{S}(\mathrm{X}) \succeq' \mathcal{M} \quad \mathcal{S} \succeq' \mathcal{S}'}{\mathcal{S} \succeq' \mathrm{X} : \mathcal{M}, \mathcal{S}'} \qquad (\succeq' \text{-4})$$

Functor Enrichment $\boxed{\mathcal{F} \succeq' \mathcal{F}'}$

$$\frac{\begin{array}{ll} \mathcal{M}_Q \succeq' \varphi(\mathcal{M}_P) & \varphi(\mathcal{M}'_P) \succeq' \mathcal{M}'_Q \\ \mathrm{Dom}(\varphi) = P & Q \cap \mathrm{FV}(\forall P.\mathcal{M}_P \to \mathcal{M}'_P) = \emptyset \end{array}}{\forall P.\mathcal{M}_P \to \mathcal{M}'_P \succeq' \forall Q.\mathcal{M}_Q \to \mathcal{M}'_Q} \qquad (\succeq' \text{-5})$$

Module Enrichment $\boxed{\mathcal{M} \succeq' \mathcal{M}'}$

$$\frac{\mathcal{S} \succeq' \mathcal{S}'}{\mathcal{S} \succeq' \mathcal{S}'} \qquad (\succeq' \text{-6})$$

$$\frac{\mathcal{F} \succeq' \mathcal{F}'}{\mathcal{F} \succeq' \mathcal{F}'} \qquad (\succeq' \text{-7})$$

Figure 5.22: An slightly different definition of enrichment with a stronger induction principle. One can show $\mathcal{O} \succeq' \mathcal{O}'$ whenever $\mathcal{O} \succeq \mathcal{O}'$.

We will also need the following (easy) lemma:

Lemma 5.36 (Filtering). *Suppose τ is of the form $\tau \equiv \varphi\left(\alpha\ \beta_0 \cdots \beta_{n-1}\right)$ (for some $n \geq 0$).If $\beta_i \notin \mathrm{Inv}(\varphi) \cup \mathrm{FV}(\alpha\ \beta_0 \cdots \beta_{i-1})$ (for all $i \in [n]$), then we must have*

$$\varphi(\alpha) = \Lambda\beta_0 \cdots \beta_{n-1}.\tau$$

(up to η, α-equivalence).

Proof (Completeness). *We can now prove the stronger theorem:*

$$\begin{array}{l}
\mathcal{O} \succeq' \mathcal{O}' \supset \\
\quad \forall \bar{\mathcal{O}}, \varphi, P, R. \\
\quad\quad \mathrm{Reg}(\varphi) \cap R = \emptyset \supset \\
\quad\quad\quad \mathcal{O}' = \varphi\left(\bar{\mathcal{O}}\right) \supset \\
\quad\quad\quad\quad \forall P \cup R \vdash \mathcal{O}\ \mathbf{Gnd} \supset \\
\quad\quad\quad\quad\quad \forall P.\exists \mathrm{Dom}(\varphi).\forall R \vdash \bar{\mathcal{O}}\ \mathbf{Slv} \supset \\
\quad\quad\quad\quad\quad\quad \left(\forall P.\forall R \vdash \mathcal{O} \succeq \bar{\mathcal{O}} \downarrow \varphi \quad \wedge \quad \mathrm{Reg}(\varphi) \subseteq P\right)
\end{array}$$

Note that we need to do induction on the relation $\succeq'$ rather than $\succeq$. Completeness then follows easily by an appeal to Lemma 5.35 (Strong Induction).

The proof itself is a tricky rule induction and requires appeals to Lemmas 5.28 (Closure), 5.31 (Grounding), 5.29 (Strengthening) and 5.30 (Weakening) and 5.36 (Filtering).

$\boxed{\succeq'\ \textbf{-1}}$ *Easy.*

$\boxed{\succeq'\ \textbf{-2}}$ *By induction we may assume:*

$$\mathrm{t} \in \mathrm{Dom}(\mathcal{S}), \tag{1}$$

$$\mathcal{S}(\mathrm{t}) = \tau, \tag{2}$$

$$\begin{array}{l}
\forall \bar{\mathcal{S}}, \varphi, P, R. \\
\quad \mathrm{Reg}(\varphi) \cap R = \emptyset \supset \\
\quad\quad \mathcal{S}' = \varphi\left(\bar{\mathcal{S}}\right) \supset \\
\quad\quad\quad \forall P \cup R \vdash \mathcal{S}\ \mathbf{Gnd} \supset \\
\quad\quad\quad\quad \forall P.\exists \mathrm{Dom}(\varphi).\forall R \vdash \bar{\mathcal{S}}\ \mathbf{Slv} \supset \\
\quad\quad\quad\quad\quad \left(\forall P.\forall R \vdash \mathcal{S} \succeq \bar{\mathcal{S}} \downarrow \varphi \quad \wedge \quad \mathrm{Reg}(\varphi) \subseteq P\right)
\end{array} \tag{3}$$

We need to show:

$$\begin{array}{l}
\forall \bar{\mathcal{S}}, \varphi, P, R. \\
\quad \mathrm{Reg}(\varphi) \cap R = \emptyset \supset \\
\quad\quad (\mathrm{t} = \tau, \mathcal{S}') = \varphi\left(\bar{\mathcal{S}}\right) \supset \\
\quad\quad\quad \forall P \cup R \vdash \mathcal{S}\ \mathbf{Gnd} \supset \\
\quad\quad\quad\quad \forall P.\exists \mathrm{Dom}(\varphi).\forall R \vdash \bar{\mathcal{S}}\ \mathbf{Slv} \supset \\
\quad\quad\quad\quad\quad \big(\forall P.\forall R \vdash \mathcal{S} \succeq \bar{\mathcal{S}} \downarrow \varphi \quad\wedge\quad \mathrm{Reg}(\varphi) \subseteq P\big)
\end{array}$$

Consider arbitrary $\bar{\mathcal{S}}$, φ, P, R *such that:*

$$\mathrm{Reg}(\varphi) \cap R = \emptyset, \tag{4}$$

$$(\mathrm{t} = \tau, \mathcal{S}') = \varphi\left(\bar{\mathcal{S}}\right), \tag{5}$$

$$\forall P \cup R \vdash \mathcal{S}\ \mathbf{Gnd}, \tag{6}$$

$$\forall P.\exists \mathrm{Dom}(\varphi).\forall R \vdash \bar{\mathcal{S}}\ \mathbf{Slv}. \tag{7}$$

By (5) *there must be some type* $\bar{\tau}$ *and structure* $\bar{\mathcal{S}}'$ *such that:*

$$\tau = \varphi\left(\bar{\tau}\right) \tag{8}$$

and

$$\mathcal{S}' = \varphi\left(\bar{\mathcal{S}}'\right), \tag{9}$$

where

$$\bar{\mathcal{S}} \ = (\mathrm{t} = \bar{\tau}, \bar{\mathcal{S}}'). \tag{10}$$

Inverting (7) *we have two cases:*

$\boxed{\mathbf{A}}$ *Assumption* (7) *was derived by S-2. Then the following premises must hold:*

$$\mathrm{FV}(\bar{\tau}) \subseteq P \cup R, \tag{a}$$

$$\forall P.\exists \mathrm{Dom}(\varphi).\forall R \vdash \bar{\mathcal{S}}'\ \mathbf{Slv}. \tag{b}$$

Lemma 5.28 (Closure) on (7) *ensures* $\mathrm{Dom}(\varphi) \cap (P \cup R) = \emptyset$. *Hence* $\varphi\left(\bar{\tau}\right) = \bar{\tau}$ *by* (a), *and equations* (2) *and* (8) *yield:*

$$\mathcal{S}(\mathrm{t}) = \bar{\tau}. \tag{c}$$

Induction hypothesis (3) *on* $\bar{\mathcal{S}}'$, φ, P *and* R, *applied to* (4),(9), (6) *and* (b) *yields:*

$$\forall P.\forall R \vdash \mathcal{S} \succeq \bar{\mathcal{S}}' \downarrow \varphi. \tag{d}$$

$$\mathrm{Reg}(\varphi) \subseteq P, \tag{e}$$

Rule (M-2) *applied to* (1), (c) *and* (d) *derives:*

$$\forall P.\forall R \vdash \mathcal{S} \succeq \mathrm{t} = \bar{\tau}, \bar{\mathcal{S}}' \downarrow \varphi. \tag{f}$$

Combining (f) *and* (e) *gives the desired result.*

$\boxed{\mathbf{B}}$ *Assumption* (7) *was derived by* *S-3*. *Then the following premises must hold:*

$$\alpha \notin P, \tag{a}$$

$$\forall i \in [n].\beta_i \in R, \tag{b}$$

$$\forall i \neq j \in [n].\beta_i \neq \beta_j, \tag{c}$$

$$\forall P \cup \{\alpha\}.\exists Q.\forall R \vdash \bar{\mathcal{S}}' \ \mathbf{Slv}, \tag{d}$$

for some α, Q, n, *and* β_i $(i \in [n])$, *where*

$$\mathrm{Dom}(\varphi) = \{\alpha\} \cup Q \tag{e}$$

and

$$\bar{\tau} = \alpha\ \beta_0 \cdots \beta_{n-1}. \tag{f}$$

Lemma 5.28 *(Closure) on* (d) *together with* (a) *ensures*

$$\alpha \notin P \cup R. \tag{g}$$

By (2), (8) *and* (f) *we have*

$$\mathcal{S}(\mathrm{t}) = \varphi\left(\alpha\ \beta_0 \cdots \beta_{n-1}\right). \tag{h}$$

Since they are equal, $\mathcal{S}(\mathrm{t})$ *and* $\alpha\ \beta_0 \cdots \beta_{n-1}$ *must be of the same kind* κ, *for some* κ, *that is:*

$$\mathcal{S}(\mathrm{t}), (\alpha\ \beta_0 \cdots \beta_{n-1}) \in \mathit{Typ}^{\kappa}. \tag{i}$$

Lemma 5.28 (Closure) on (7), ensures

$$\mathrm{Dom}(\varphi) \cap R = \emptyset, \tag{j}$$

$$P \cap R = \emptyset. \tag{k}$$

Consider arbitrary $i \in [n]$. Then $\beta_i \in R$ by (b) from which $\beta_i \notin \mathrm{Inv}(\varphi)$ follows by (j) and (4); moreover, from (g) we have $\beta_i \neq \alpha$, and from (c) we obtain $\beta_i \notin \{\beta_j \mid j \in i-1\}$. Hence $\beta_i \notin \mathrm{Inv}(\varphi) \cup \mathrm{FV}(\alpha\ \beta_0 \cdots \beta_{i-1})$. Since i was arbitrary, we have:

$$\forall i \in [n].\beta_i \notin \mathrm{FV}(\alpha\ \beta_0 \cdots \beta_{i-1}). \tag{l}$$

Lemma 5.36 (Filtering) on equation (h) and (l) determines that:

$$\varphi(\alpha) = \Lambda\beta_0 \cdots \beta_{n-1}.\mathcal{S}(\mathrm{t}). \tag{m}$$

From (m) we clearly have: $\mathrm{FV}(\mathcal{S}(\mathrm{t})) \setminus \{\beta_i \mid i \in [n]\} \subseteq \mathrm{FV}(\varphi(\alpha))$. *Hence* $\mathrm{FV}(\mathcal{S}(\mathrm{t})) \subseteq \mathrm{FV}(\varphi(\alpha)) \cup \{\beta_i \mid i \in [n]\}$. *Now*

$$\begin{aligned} \mathrm{FV}(\mathcal{S}(\mathrm{t})) \cap R &\subseteq (\mathrm{FV}(\varphi(\alpha)) \cup \{\beta_i \mid i \in [n]\}) \cap R \\ &\subseteq (\mathrm{FV}(\varphi(\alpha)) \cap R) \cup (\{\beta_i \mid i \in [n]\} \cap R). \end{aligned}$$

Hence by (e), (4) and (b):

$$\mathrm{FV}(\mathcal{S}(\mathrm{t})) \cap R \subseteq \{\beta_i \mid i \in [n]\}. \tag{n}$$

Inverting (6) we have, in particular by (1), $\mathrm{FV}(\mathcal{S}(\mathrm{t})) \subseteq P \cup R$, *which, together with (k) and (n), lets us show:*

$$\mathrm{FV}(\Lambda\beta_0 \cdots \beta_{n-1}.\mathcal{S}(\mathrm{t})) \subseteq P. \tag{o}$$

Lemma 5.29 (Strengthening) (d) on α and $\Lambda\beta_0 \cdots \beta_{n-1}.\mathcal{S}(\mathrm{t})$ using (a) and (o) produces:

$$\forall P.\exists Q.\forall R \vdash [\Lambda\beta_0 \cdots \beta_{n-1}.\mathcal{S}(\mathrm{t})/\alpha]\left(\bar{\mathcal{S}}'\right) \mathbf{Slv}. \tag{p}$$

Lemma 5.28 (Closure) on (d) yields:

$$\alpha \notin Q, \tag{q}$$

$$Q \cap P = \emptyset. \tag{r}$$

Let $\varphi' \overset{\text{def}}{=} \{\alpha \mapsto \varphi(\alpha) \,|\, \alpha \in Q\}$. *Then* (m) *and* (q) *give:*

$$\varphi = [\Lambda\beta_0 \cdots \beta_{n-1}.\mathcal{S}(\mathrm{t})/\alpha] \mid \varphi', \tag{s}$$

since

$$\mathrm{Dom}(\varphi') = Q. \tag{t}$$

In particular, by (4) *we also have:*

$$\mathrm{Reg}(\varphi') \cap R = \emptyset. \tag{u}$$

From (r), (q) *and* (o) *it is easy to verify that* $\mathrm{Dom}(\varphi) \cap \mathrm{Inv}([\Lambda\beta_0 \cdots \beta_{n-1}.\mathcal{S}(\mathrm{t})/\alpha]) = \emptyset$. *Hence, by* (9) *we have:*

$$\mathcal{S}' = \varphi'\left([\Lambda\beta_0 \cdots \beta_{n-1}.\mathcal{S}(\mathrm{t})/\alpha]\left(\bar{\mathcal{S}}'\right)\right). \tag{v}$$

Induction hypothesis (3) *on* $[\Lambda\beta_0 \cdots \beta_{n-1}.\mathcal{S}(\mathrm{t})/\alpha]\left(\bar{\mathcal{S}}'\right)$, φ', P *and* R, *applied to* (u), (v), (6) *and* (p) *(using equality* (t)*) yields:*

$$\forall P.\forall R \vdash \mathcal{S} \succeq [\Lambda\beta_0 \cdots \beta_{n-1}.\mathcal{S}(\mathrm{t})/\alpha]\left(\bar{\mathcal{S}}'\right) \downarrow \varphi', \tag{w}$$

$$\mathrm{Reg}(\varphi') \subseteq P. \tag{x}$$

Rule (*M*-3) *applied to* (g), (1), (i), (n) *and* (w) *derives:*

$$\forall P.\forall R \vdash \mathcal{S} \succeq \mathrm{t} = \alpha\ \beta_0 \cdots \beta_{n-1}, \bar{\mathcal{S}}' \downarrow ([\Lambda\beta_0 \cdots \beta_{n-1}.\mathcal{S}(\mathrm{t})/\alpha] \mid \varphi')$$

i.e.:

$$\forall P.\forall R \vdash \mathcal{S} \succeq \bar{\mathcal{S}} \downarrow \varphi. \tag{y}$$

by (10) *and* (s).
Finally, using (s), (o) *and* (x) *it is easy to verify*

$$\begin{aligned} \mathrm{Reg}(\varphi) &= \mathrm{Reg}([\Lambda\beta_0 \cdots \beta_{n-1}.\mathcal{S}(\mathrm{t})/\alpha] \mid \varphi') \\ &= \mathrm{FV}(\Lambda\beta_0 \cdots \beta_{n-1}.\mathcal{S}(\mathrm{t})) \cup \mathrm{Reg}(\varphi') \\ &\subseteq P. \end{aligned} \tag{z}$$

Combining (y) *and* (z) *gives the desired result.*

$\boxed{\succeq' \textbf{-3}}$ *Similar to case* $\boxed{\succeq' \textbf{-2}}$, *reasoning as in sub-case* $\boxed{\boldsymbol{A}}$.

$\boxed{\succeq' \textbf{-4}}$ *By induction we may assume:*

$$\mathrm{X} \in \mathrm{Dom}(\mathcal{S}), \tag{1}$$

$$\begin{array}{l}
\forall \bar{\mathcal{M}}, \varphi, P, R. \\
\quad \mathrm{Reg}(\varphi) \cap R = \emptyset \supset \\
\quad\quad \mathcal{M} = \varphi\left(\bar{\mathcal{M}}\right) \supset \\
\quad\quad\quad \forall P \cup R \vdash \mathcal{S}(\mathrm{X})\ \textbf{Gnd} \supset \\
\quad\quad\quad\quad \forall P.\exists \mathrm{Dom}(\varphi).\forall R \vdash \bar{\mathcal{M}}\ \textbf{Slv} \supset \\
\quad\quad\quad\quad\quad \big(\forall P.\forall R \vdash \mathcal{S}(\mathrm{X}) \succeq \bar{\mathcal{M}} \downarrow \varphi \ \wedge\ \mathrm{Reg}(\varphi) \subseteq P\big)
\end{array} \tag{2}$$

$$\begin{array}{l}
\forall \bar{\mathcal{S}}, \varphi, P, R. \\
\quad \mathrm{Reg}(\varphi) \cap R = \emptyset \supset \\
\quad\quad \mathcal{S}' = \varphi\left(\bar{\mathcal{S}}\right) \supset \\
\quad\quad\quad \forall P \cup R \vdash \mathcal{S}\ \textbf{Gnd} \supset \\
\quad\quad\quad\quad \forall P.\exists \mathrm{Dom}(\varphi).\forall R \vdash \bar{\mathcal{S}}\ \textbf{Slv} \supset \\
\quad\quad\quad\quad\quad \big(\forall P.\forall R \vdash \mathcal{S} \succeq \bar{\mathcal{S}} \downarrow \varphi \ \wedge\ \mathrm{Reg}(\varphi) \subseteq P\big)
\end{array} \tag{3}$$

We need to show:

$$\begin{array}{l}
\forall \bar{\mathcal{S}}, \varphi, P, R. \\
\quad \mathrm{Reg}(\varphi) \cap R = \emptyset \supset \\
\quad\quad \mathrm{X} : \mathcal{M}, \mathcal{S}' = \varphi\left(\bar{\mathcal{S}}\right) \supset \\
\quad\quad\quad \forall P \cup R \vdash \mathcal{S}\ \textbf{Gnd} \supset \\
\quad\quad\quad\quad \forall P.\exists \mathrm{Dom}(\varphi).\forall R \vdash \bar{\mathcal{S}}\ \textbf{Slv} \supset \\
\quad\quad\quad\quad\quad \big(\forall P.\forall R \vdash \mathcal{S} \succeq \bar{\mathcal{S}} \downarrow \varphi \ \wedge\ \mathrm{Reg}(\varphi) \subseteq P\big)
\end{array}$$

Consider arbitrary $\bar{\mathcal{S}}$, φ, P, R *such that:*

$$\mathrm{Reg}(\varphi) \cap R = \emptyset, \tag{4}$$

$$\mathrm{X} : \mathcal{M}, \mathcal{S}' = \varphi\left(\bar{\mathcal{S}}\right), \tag{5}$$

$$\forall P \cup R \vdash \mathcal{S}\ \textbf{Gnd}, \tag{6}$$

$$\forall P.\exists \mathrm{Dom}(\varphi).\forall R \vdash \bar{\mathcal{S}}\ \textbf{Slv}. \tag{7}$$

By (5) *there must be some module* $\bar{\mathcal{M}}$ *and structure* $\bar{\mathcal{S}}'$ *such that:*

$$\mathcal{M} = \varphi\left(\bar{\mathcal{M}}\right), \tag{8}$$

$$\mathcal{S}' = \varphi\left(\bar{\mathcal{S}}'\right), \tag{9}$$

where

$$\bar{\mathcal{S}} \ = \mathrm{X} : \bar{\mathcal{M}}, \bar{\mathcal{S}}'. \tag{10}$$

Inverting (7) *the following premises must hold:*

$$\forall P.\exists Q_1.\forall R \vdash \bar{\mathcal{M}}\ \mathbf{Slv}, \tag{11}$$

$$\forall P \cup Q_1.\exists Q_2.\forall R \vdash \bar{\mathcal{S}}'\ \mathbf{Slv}, \tag{12}$$

for some Q_1 *and* Q_2 *with*

$$\mathrm{Dom}(\varphi) = Q_1 \cup Q_2. \tag{13}$$

Two applications of Lemma 5.28 (Closure) to (11) *and* (12) *yield:*

$$\mathrm{FV}(\bar{\mathcal{M}}) \subseteq P \cup Q_1 \cup R, \tag{14}$$

$$Q_1 \cap P = \emptyset, \tag{15}$$

$$Q_2 \cap (P \cup Q_1 \cup R) = \emptyset. \tag{16}$$

Let $\varphi_1 \overset{\text{def}}{=} \{\alpha \mapsto \varphi(\alpha) \,|\, \alpha \in Q_1\}$ *and* $\varphi_2 \overset{\text{def}}{=} \{\alpha \mapsto \varphi(\alpha) \,|\, \alpha \in Q_2\}$. *Then*

$$\mathrm{Dom}(\varphi_1) = Q_1, \tag{17}$$

$$\mathrm{Dom}(\varphi_2) = Q_2 \tag{18}$$

and

$$\varphi = \varphi_1 \mid \varphi_2, \tag{19}$$

by (13) *and* (16). *Moreover, from* (4) *we obtain:*

$$\mathrm{Reg}(\varphi_1) \cap R = \emptyset, \tag{20}$$

$$\mathrm{Reg}(\varphi_2) \cap R = \emptyset. \tag{21}$$

Now $\mathrm{Dom}(\varphi_2) \cap \mathrm{FV}(\bar{\mathcal{M}}) = \emptyset$ *follows from* (14), (16) *and* (18). *Hence* $\varphi\left(\bar{\mathcal{M}}\right) = (\varphi_1 \mid \varphi_2)\left(\bar{\mathcal{M}}\right) = \varphi_1\left(\bar{\mathcal{M}}\right)$, *and by* (8):

$$\mathcal{M} = \varphi_1\left(\bar{\mathcal{M}}\right). \tag{22}$$

Inverting (6) *we have, in particular by* (1),

$$\forall P \cup R \vdash \mathcal{S}(\mathrm{X})\ \mathbf{Gnd}. \tag{23}$$

Induction hypothesis (3) *on* $\bar{\mathcal{M}}$, φ_1, P *and* R, *applied to* (20), (22), (23) *and* (11) *(using equality* (17)*) yields:*

$$\forall P.\forall R \vdash \mathcal{S}(\mathrm{X}) \succeq \bar{\mathcal{M}}\ \downarrow\ \varphi_1, \tag{24}$$

$$\mathrm{Reg}(\varphi_1) \subseteq P. \tag{25}$$

It is easy to verify that $\mathrm{Dom}(\varphi_2) \cap \mathrm{Inv}(\varphi_1) = \emptyset$.
Hence $\varphi\left(\bar{\mathcal{S}}'\right) = (\varphi_1 \mid \varphi_2)\left(\bar{\mathcal{S}}'\right) = \varphi_2\left(\varphi_1\left(\bar{\mathcal{S}}'\right)\right)$, *and by* (9) *we have:*

$$\mathcal{S}' = \varphi_2\left(\varphi_1\left(\bar{\mathcal{S}}'\right)\right). \tag{26}$$

Lemma 5.29 (Strengthening) on (12) *with* φ_1 *using* (15), (17) *and* (25) *produces:*

$$\forall P.\exists Q_2.\forall R \vdash \varphi_1\left(\bar{\mathcal{S}}'\right)\ \mathbf{Slv}. \tag{27}$$

Induction hypothesis (3) *on* $\varphi_1\left(\bar{\mathcal{S}}'\right)$, φ_2, P *and* R, *applied to* (21), (26), (6) *and* (27) *(using equality* (18)*) yields:*

$$\forall P.\forall R \vdash \mathcal{S} \succeq \varphi_1\left(\bar{\mathcal{S}}'\right)\ \downarrow\ \varphi_2, \tag{28}$$

$$\mathrm{Reg}(\varphi_2) \subseteq P. \tag{29}$$

Rule (*M*-5) *applied to* (1), (24), *and* (28) *derives*

$$\forall P.\forall R \vdash \mathcal{S} \succeq \mathrm{X} : \bar{\mathcal{M}}, \bar{\mathcal{S}}'\ \downarrow\ (\varphi_1 \mid \varphi_2),$$

i.e. using equations (10) *and* (19):

$$\forall P.\forall R \vdash \mathcal{S} \succeq \bar{\mathcal{S}}\ \downarrow\ \varphi. \tag{30}$$

Using (19), (25) *and* (29) *it is easy to verify:*

$$\begin{array}{rcl}\mathrm{Reg}(\varphi) & = & \mathrm{Reg}(\varphi_1 \mid \varphi_2) \\ & = & \mathrm{Reg}(\varphi_1) \cup \mathrm{Reg}(\varphi_2) \\ & \subseteq & P. \end{array} \tag{31}$$

Combining (30) *and* (31) *gives the desired result.*

$\boxed{\succeq' \textbf{-5}}$ *By induction we may assume:*

$$\begin{array}{l}\forall \bar{\mathcal{M}}, \varphi, P, R. \\ \quad \mathrm{Reg}(\varphi) \cap R = \emptyset \supset \\ \quad\quad \varphi'(\mathcal{M}_N) = \varphi(\bar{\mathcal{M}}) \supset \\ \quad\quad\quad \forall P \cup R \vdash \mathcal{M}_M \ \textbf{Gnd} \supset \\ \quad\quad\quad\quad \forall P.\exists \mathrm{Dom}(\varphi).\forall R \vdash \bar{\mathcal{M}} \ \textbf{Slv} \supset \\ \quad\quad\quad\quad\quad \big(\forall P.\forall R \vdash \mathcal{M}_M \succeq \bar{\mathcal{M}} \downarrow \varphi \ \ \wedge \ \ \mathrm{Reg}(\varphi) \subseteq P\big)\end{array} \tag{1}$$

$$\begin{array}{l}\forall \bar{\mathcal{M}}, \varphi, P, R. \\ \quad \mathrm{Reg}(\varphi) \cap R = \emptyset \supset \\ \quad\quad \mathcal{M}'_M = \varphi(\bar{\mathcal{M}}) \supset \\ \quad\quad\quad \forall P \cup R \vdash \varphi'(\mathcal{M}'_N) \ \textbf{Gnd} \supset \\ \quad\quad\quad\quad \forall P.\exists \mathrm{Dom}(\varphi).\forall R \vdash \bar{\mathcal{M}} \ \textbf{Slv} \supset \\ \quad\quad\quad\quad\quad \big(\forall P.\forall R \vdash \varphi'(\mathcal{M}'_N) \succeq \bar{\mathcal{M}} \downarrow \varphi \ \ \wedge \ \ \mathrm{Reg}(\varphi) \subseteq P\big)\end{array} \tag{2}$$

$$\mathrm{Dom}(\varphi') = N, \tag{3}$$

$$M \cap \mathrm{FV}(\forall N.\mathcal{M}_N \to \mathcal{M}'_N) = \emptyset. \tag{4}$$

We need to show:

$$\begin{array}{l}\forall \bar{\mathcal{F}}, \varphi, P, R. \\ \quad \mathrm{Reg}(\varphi) \cap R = \emptyset \supset \\ \quad\quad \forall M.\mathcal{M}_M \to \mathcal{M}'_M = \varphi(\bar{\mathcal{F}}) \supset \\ \quad\quad\quad \forall P \cup R \vdash \forall N.\mathcal{M}_N \to \mathcal{M}'_N \ \textbf{Gnd} \supset \\ \quad\quad\quad\quad \forall P.\exists \mathrm{Dom}(\varphi).\forall R \vdash \bar{\mathcal{F}} \ \textbf{Slv} \supset \\ \quad\quad\quad\quad\quad \big(\forall P.\forall R \vdash \forall N.\mathcal{M}_N \to \mathcal{M}'_N \succeq \bar{\mathcal{F}} \downarrow \varphi \ \ \wedge \ \ \mathrm{Reg}(\varphi) \subseteq P\big)\end{array}$$

Consider arbitrary $\bar{\mathcal{F}}$, φ, P, R *such that:*

$$\mathrm{Reg}(\varphi) \cap R = \emptyset, \tag{5}$$

$$\forall M.\mathcal{M}_M \rightarrow \mathcal{M}'_M = \varphi\left(\bar{\mathcal{F}}\right), \tag{6}$$

$$\forall P \cup R \vdash \forall N.\mathcal{M}_N \rightarrow \mathcal{M}'_N \ \mathbf{Gnd}, \tag{7}$$

$$\forall P.\exists \mathrm{Dom}(\varphi).\forall R \vdash \bar{\mathcal{F}} \ \mathbf{Slv}. \tag{8}$$

W.l.o.g. we may assume (by renaming bound variables if necessary) that:

$$M \cap \mathrm{Inv}(\varphi) = \emptyset, \tag{9}$$

$$M \cap N = \emptyset. \tag{10}$$

Then, by (6), can assume:

$$\bar{\mathcal{F}} \equiv \forall M.\bar{\mathcal{M}}_M \rightarrow \bar{\mathcal{M}}'_M. \tag{11}$$

for some $\bar{\mathcal{M}}_M$ and $\bar{\mathcal{M}}'_M$.

Moreover, by (9) and (11) we have

$$\varphi\left(\bar{\mathcal{F}}\right) = \forall M.\varphi\left(\bar{\mathcal{M}}_M\right) \rightarrow \varphi\left(\bar{\mathcal{M}}'_M\right), \tag{12}$$

where, by (6):

$$\mathcal{M}_M = \varphi\left(\bar{\mathcal{M}}_M\right), \tag{13}$$

$$\mathcal{M}'_M = \varphi\left(\bar{\mathcal{M}}'_M\right). \tag{14}$$

Inverting (7) the following premises must hold:

$$\forall P \cup R.\exists N.\forall \emptyset \vdash \mathcal{M}_N \ \mathbf{Slv}, \tag{15}$$

$$\forall P \cup R \cup N \vdash \mathcal{M}'_N \ \mathbf{Gnd}. \tag{16}$$

Similarly, inverting (8), using (11), the following premises must also hold:

$$\forall P \cup R.\exists M.\forall \emptyset \vdash \bar{\mathcal{M}}_M \ \mathbf{Slv}, \tag{17}$$

$$\forall P.\exists \mathrm{Dom}(\varphi).\forall R \cup M \vdash \bar{\mathcal{M}}'_M \ \mathbf{Slv}. \tag{18}$$

Two applications of Lemma 5.28 (Closure) to (17) and (18) yield $\mathrm{FV}(\bar{\mathcal{M}}_M) \subseteq P \cup R \cup M$ *and* $\mathrm{Dom}(\varphi) \cap (P \cup R \cup M) = \emptyset$*. Hence* $\mathrm{Dom}(\varphi) \cap \mathrm{FV}(\bar{\mathcal{M}}_M) = \emptyset$, $\varphi\left(\bar{\mathcal{M}}_M\right) = \bar{\mathcal{M}}_M$ *and, by (13), we have:*

$$\mathcal{M}_M = \bar{\mathcal{M}}_M. \tag{19}$$

By Lemma 5.31 (Grounding) on (17) using equation (19) we obtain:

$$\forall P \cup R \cup M \vdash \mathcal{M}_M \ \mathbf{Gnd}. \tag{20}$$

Lemma 5.30 (Weakening) on (15) with M*, using assumption (10) provides:*

$$\forall P \cup R \cup M.\exists N.\forall \emptyset \vdash \mathcal{M}_N \ \mathbf{Slv}. \tag{21}$$

Clearly

$$\mathrm{Reg}(\varphi') \cap \emptyset = \emptyset, \tag{22}$$

$$\varphi'\left(\mathcal{M}_N\right) = \varphi'\left(\mathcal{M}_N\right). \tag{23}$$

Induction hypothesis (1) on $\mathcal{M}_N$, φ', $P \cup R \cup M$ *and* $\emptyset$*, applied to (22), (23), (20) and (21) (using equation (3)) yields*

$$\forall P \cup R \cup M.\forall \emptyset \vdash \bar{\mathcal{M}}_M \succeq \mathcal{M}_N \ \downarrow \ \varphi', \tag{24}$$

$$\mathrm{Reg}(\varphi') \subseteq P \cup R \cup M. \tag{25}$$

Lemma 5.30 (Weakening) on (16) with M *provides:*

$$\forall P \cup R \cup N \cup M \vdash \mathcal{M}'_N \ \mathbf{Gnd}. \tag{26}$$

Lemma 5.28 (Closure) on (15) establishes $N \cap (P \cup R) = \emptyset$*, which together with (10) yields:*

$$N \cap (P \cup R \cup M) = \emptyset. \tag{27}$$

Lemma 5.29 (Strengthening) on (26) with φ' *using (3), (27) and (25) yields:*

$$\forall P \cup R \cup M \vdash \varphi'\left(\mathcal{M}'_N\right) \ \mathbf{Gnd}. \tag{28}$$

From (9)*, we have, in particular,* $M \cap \mathrm{Reg}(\varphi) = \emptyset$*, which, together with* (5)*, ensures:*

$$\mathrm{Reg}(\varphi) \cap (R \cup M) = \emptyset. \tag{29}$$

Induction hypothesis (2) *on* $\bar{\mathcal{M}}'_M$*,* φ*,* P *and* $R \cup M$*, applied to* (29)*,* (14)*,* (28) *and* (18) *yields*

$$\forall P.\forall R \cup M \vdash \varphi'\left(\mathcal{M}'_N\right) \succeq \bar{\mathcal{M}}'_M \downarrow \varphi, \tag{30}$$

$$\mathrm{Reg}(\varphi) \subseteq P. \tag{31}$$

Lemma 5.28 (Closure) on (17) *establishes:*

$$M \cap (P \cup R) = \emptyset. \tag{32}$$

Applying Rule (M-6) *to* (27)*,* (32)*,* (24) *and* (30) *derives*

$$\forall P.\forall R \vdash \forall N.\mathcal{M}_N \to \mathcal{M}'_N \succeq \forall M.\bar{\mathcal{M}}_M \to \bar{\mathcal{M}}'_M \downarrow \varphi,$$

which, by equation (11)*, may be rewritten as:*

$$\forall P.\forall R \vdash \forall N.\mathcal{M}_N \to \mathcal{M}'_N \succeq \bar{\mathcal{F}} \downarrow \varphi. \tag{33}$$

Combining (33) *and* (31) *gives the desired result.*

$\boxed{\succeq' \textbf{-6}}$ *Trivial induction.*

$\boxed{\succeq' \textbf{-7}}$ *Trivial induction.*

5.7 A Type Checking Algorithm for Higher-Order Modules

In the preceding sections we focussed on producing an algorithm for matching. Although the algorithm is only sound and complete provided its inputs are ground and solvable, we can now turn to the static semantics of Higher-Order Modules to show that, whenever we need to invoke the algorithm, the matching problem will indeed be well-posed. To this end, we first define a notion of ground contexts, ground existential modules, and solvable signatures.

Ground Contexts $\boxed{\vdash \mathcal{C}\ \mathbf{Gnd}}$

$$\frac{\forall \mathrm{X} \in \mathrm{Dom}(\mathcal{C}).\ \forall \mathrm{FV}(\mathcal{C}(\mathrm{X})) \vdash \mathcal{C}(\mathrm{X})\ \mathbf{Gnd}}{\vdash \mathcal{C}\ \mathbf{Gnd}} \qquad (G\text{-}5)$$

Ground Existential Modules $\boxed{\vdash \mathcal{X}\ \mathbf{Gnd}}$

$$\frac{\forall \mathrm{FV}(\exists P.\mathcal{M}) \vdash \mathcal{M}\ \mathbf{Gnd}}{\vdash \exists P.\mathcal{M}\ \mathbf{Gnd}} \qquad (G\text{-}6)$$

Solvable Signatures $\boxed{\vdash \mathcal{L}\ \mathbf{Slv}}$

$$\frac{\forall \mathrm{FV}(\Lambda P.\mathcal{M}).\exists P.\forall \emptyset \vdash \mathcal{M}\ \mathbf{Slv}}{\vdash \Lambda P.\mathcal{M}\ \mathbf{Slv}} \qquad (G\text{-}7)$$

Figure 5.23: The definition of ground contexts, ground existential modules, and solvable signatures.

Definition 5.37 (Well-formed Contexts, Existential Modules, Signatures). The three predicates $\vdash \mathcal{C}$ **Gnd**, $\vdash \mathcal{X}$ **Gnd** and $\vdash \mathcal{L}$ **Slv** are defined by the rules in Figure 5.23.

We will need a lemma that allows us to eliminate spurious variables from judgements stating solvability and groundedness:

Lemma 5.38 (Elimination).

- *If* $\forall P \vdash \mathcal{O}$ **Gnd** *then* $\forall \mathrm{FV}(\mathcal{O}) \vdash \mathcal{O}$ **Gnd**.
- *If* $\forall P.\exists Q.\forall R \vdash \mathcal{O}$ **Slv** *then* $\forall P \cap \mathrm{FV}(\mathcal{O}).\exists Q.\forall R \vdash \mathcal{O}$ **Slv**.

Proof. *The proof proceeds by strong rule induction. The proof is easy and requires appeals to Lemmas 5.28 (Closure) and 5.29 (Strengthening).*

To show the well-formedness of semantic functors introduced by Rules (H-7) and (H-18) we will also need the following lemma:

Lemma 5.39 (Raising). *If* $\forall P \cup P'.\exists Q.\forall R \vdash \mathcal{O}$ **Slv**, *where*

- $P \cap P' = \emptyset$,
- $P' = \{\alpha_0^{\kappa_0}, \ldots, \alpha_{n-1}^{\kappa_{n-1}}\}$,
- $Q' = \{\beta^{\kappa_0 \to \cdots \kappa_{n-1} \to \kappa} | \beta^\kappa \in Q\}$,

- $[Q'/Q] = \{\beta^\kappa \mapsto \beta^{\kappa_0 \to \cdots \kappa_{n-1} \to \kappa}\ \alpha_0 \cdots \alpha_{n-1} | \beta^\kappa \in Q\}$, *and*
- $Q' \cap (P \cup P' \cup R) = \emptyset$,

then $\forall P.\exists Q'.\forall R \cup P' \vdash [Q'/Q]\,(\mathcal{O})\ \mathbf{Slv}$.

Proof. *We first prove a simpler lemma for the case where P' consists of a single variable:*

$$\begin{array}{l}\forall P.\exists Q.\forall R \vdash \mathcal{O}\ \mathbf{Slv} \supset \\ \quad \forall \alpha^\kappa \in P.\forall \beta^{\kappa'} \in Q.\beta^{\kappa \to \kappa'} \notin P \cup R \supset \\ \quad\quad \forall P \setminus \{\alpha^\kappa\}.\exists\{\beta^{\kappa\to\kappa'} | \beta^{\kappa'} \in Q\}.\forall R \cup \{\alpha^\kappa\} \vdash \mathcal{O}'\ \mathbf{Slv} \\ \quad\quad\quad \textit{where } \mathcal{O}' \equiv \{\beta^{\kappa'} \mapsto \beta^{\kappa\to\kappa'}\ \alpha^\kappa | \beta^{\kappa'} \in Q\}\,(\mathcal{O})\end{array}$$

The proof proceeds by strong rule induction.

Lemma 5.39 then follows easily by induction on n, the size of P'.

We will also need a further simple lemma to enable invocation of the algorithm in Rules (H-19), (H-20) and (H-21).

Lemma 5.40 (Invocation).

Provided $\forall \mathrm{FV}(\mathcal{M}) \vdash \mathcal{M}\ \mathbf{Gnd}$ *and* $\vdash \Lambda P.\mathcal{M}'\ \mathbf{Slv}$, *where (without loss of generality)* $P \cap \mathrm{FV}(\mathcal{M}) = \emptyset$, *we have, for any* φ:

$$\begin{array}{c}\mathrm{Dom}(\varphi) = P \textit{ and } \mathcal{M} \succeq \varphi\,(\mathcal{M}'), \\ \textit{if, and only if,} \\ \forall \mathrm{FV}(\mathcal{M}) \cup \mathrm{FV}(\Lambda P.\mathcal{M}').\forall\emptyset \vdash \mathcal{M} \succeq \mathcal{M}' \downarrow \varphi.\end{array}$$

Proof. *A simple consequence of Lemma 5.30 (Weakening), Theorem 5.32 (Soundness) and Theorem 5.33 (Completeness).*

This final lemma is needed to show that realisations cannot introduce spurious free variables:

Lemma 5.41 (Free Variables).

Provided $\forall P \vdash \mathcal{M}\ \mathbf{Gnd}$ *and* $\forall P.\exists \mathrm{Dom}(\varphi).\forall R \vdash \mathcal{O}'\ \mathbf{Slv}$, *we have* $\mathcal{O} \succeq \varphi\,(\mathcal{O}')$ *implies* $\mathrm{FV}(\mathrm{Reg}(\varphi)) \subseteq P$.

Proof. *This is a simple consequence of Strong Induction (Lemma 5.35) and the stronger statement we used to prove Completeness (Theorem 5.33).*

We can now prove the main theorem which justifies the correctness of appeals to the matching algorithm in Rules (H-19), (H-20) and H-21:

Lemma 5.42 (Invariance).
Provided $\vdash \mathcal{C}$ **Gnd**, *we have:*

- $\mathcal{C} \vdash \mathrm{d} \rhd d$ *implies* $\mathrm{FV}(d) \subseteq \mathrm{FV}(\mathcal{C})$.
- $\mathcal{C} \vdash \mathrm{v} \rhd v$ *implies* $\mathrm{FV}(v) \subseteq \mathrm{FV}(\mathcal{C})$.
- $\mathcal{C} \vdash \mathrm{e} : v$ *implies* $\mathrm{FV}(v) \subseteq \mathrm{FV}(\mathcal{C})$.
- $\mathcal{C} \vdash \mathrm{do} \rhd d$ *implies* $\mathrm{FV}(d) \subseteq \mathrm{FV}(\mathcal{C})$.
- $\mathcal{C} \vdash \mathrm{vo} : v$ *implies* $\mathrm{FV}(v) \subseteq \mathrm{FV}(\mathcal{C})$.
- $\mathcal{C} \vdash \mathrm{S} \rhd \mathcal{L}$ *implies* $\vdash \mathcal{L}$ **Slv** *and* $\mathrm{FV}(\mathcal{L}) \subseteq \mathrm{FV}(\mathcal{C})$.
- $\mathcal{C} \vdash \mathrm{B} \rhd \mathcal{L}$ *implies* $\vdash \mathcal{L}$ **Slv** *and* $\mathrm{FV}(\mathcal{L}) \subseteq \mathrm{FV}(\mathcal{C})$.
- $\mathcal{C} \vdash \mathrm{m} : \mathcal{X}$ *implies* $\vdash \mathcal{X}$ **Gnd** *and* $\mathrm{FV}(\mathcal{X}) \subseteq \mathrm{FV}(\mathcal{C})$.
- $\mathcal{C} \vdash \mathrm{b} : \mathcal{X}$ *implies* $\vdash \mathcal{X}$ **Gnd** *and* $\mathrm{FV}(\mathcal{X}) \subseteq \mathrm{FV}(\mathcal{C})$.

Proof. *The proof proceeds by induction on the rules of the static semantics. The proofs of the first three statements are Core language dependent but must be carried out simultaneously with the proofs of the remaining statements. We cannot easily factor them out as separate lemmas since we need to maintain the invariant that the context is ground in order to be able to use the fourth and fifth statements. Other than that, the proof is fairly easy. In particular, we need to appeal to Lemma 5.39 (Raising) in cases H-18 (skolemising a functor) and H-7 (parameterising a functor signature). Lemma 5.38 (Elimination) is needed in a number of cases to remove spurious bound variables in order to establish that the resulting semantic objects are indeed well-formed. In cases H-19 and H-20, Lemma 5.41 (Free Variables) is used to show that a matching realisation does not introduce any variables that did not already occur free in the current context. The fact that the existential module inferred by Rule H-21 is ground is a consequence of Lemma 5.31 (Grounding).*

5.8 Contribution and Relation to Biswas's Work

Since this chapter relies heavily on the work of Biswas [Bis95], we should make its relation to it clear. Biswas studies a skeletal Modules language supporting uniquely-kinded Core definable types but no Core value bindings, while we prove our results for a full language supporting many-kinded Core definable types, Core values and enrichment on Core value types. The clever observation that higher-order variables may be used to interpret higher-order functor signatures is due to Biswas. His presentation of the static semantics, especially of signature expressions, is very operational; ours is not. The underlying intuition, though not the precise statement, of our specification of enrichment (Specification 5.12) as a combination of polymorphic generalisation and contra-variant enrichment, is reconstructed from his informal account. Our definition of enrichment (Definition 3.17) is simpler than his operational formulation. We justify its suitability for subtyping by proving that it is a pre-order; he does not. Our matching algorithm (Algorithm 5.25), unlike his, is fully deterministic and requires less book-keeping, but it is basically a slightly optimised version of his algorithm and the credit for the underlying ideas should lie with him. Biswas sketches a proof that his algorithm is sound and complete. The proofs of soundness and completeness of the algorithm presented here are my own, although the sequence of preparatory lemmas and the definitions of ground and solvable objects owe a great debt to the lemmas and more complicated relations presented in Biswas's paper.

My contribution, in relation to Biswas's work, can be summarised as a rational reconstruction of his ideas that builds on the reformulation of the static semantics in Chapter 4, resulting in a more accessible account. Biswas's work, regrettably, does not seem to have had the significant impact on the wider Standard ML community that it deserves. I believe this is largely due to its difficult presentation. I hope the work in this chapter remedies this situation.

The novel contribution of this chapter is the addition of generativity using applicative functors, and the successful generalisation of Biswas's ideas from a skeletal to a full language, particularly to a language supporting many-kinded Core definable types (e.g. the parameterised types of Core-ML). These accomplishments address the two important areas for future research singled out in Biswas's concluding remarks. In combination, these ideas can now be used to design an acceptable, higher-order version of Standard ML's first-order modules language.

Chapter 6

Separate Compilation for Modules

In this chapter, we address the foundations for the *separate compilation* of Modules. One of the main criticisms of Standard ML Modules is its perceived lack of support for separate compilation. In Section 6.1 we set the scene by briefly describing the approach to separate compilation commonly taken in traditional programming languages. The success of this approach relies on identifying a suitable notion of compilation unit, where each unit can be factored into a description of the unit's implementation, and a description of the interface that this implementation presents to other compilation units. In Section 6.2, we review the naive approach to separate compilation in Standard ML, that attempts to identify compilation units with constrained module definitions, and show why it fails. We place the blame for this failure, not on the semantics of Modules, but on an inappropriate choice of compilation unit. In Section 6.3, we identify an alternative notion of compilation unit, based on abstracted module definitions, that satisfies the requirements of separate compilation. In Section 6.4 we reveal a theoretical sense in which our solution is only partial: a module expression may fail to admit a complete syntactic representation of its type, preventing a programmer from fully specifying its interface using an abstraction. After analysing the problem we suggest appropriate modifications to the semantics, which are formalised for a skeletal higher-order modules language in Section 6.5. The adequacy of our proposal is expressed by a theorem, whose proof is sketched. Section 6.6 concludes this chapter with a brief assessment.

6.1 Modules and Separate Compilation

Leroy gives a nice summary of modularisation and separate compilation [Ler94]:

> "*Modularisation* is the process of decomposing a program in small units (modules) that can be understood in isolation by the programmers, and making the relations between these units explicit to the programmers. *Separate compilation* is the process of decomposing a program in small units (compilation units) that can be typechecked and compiled separately by the compiler, and making the relations between these units explicit to the compiler and linker. Both processes are required for realistic programming: modularisation makes large programs understandable by programmers; separate compilation makes large programs tractable by compilers."

In the simplest separate compilation schemes, each compilation unit has a name, a public interface and a private implementation. A unit's interface records the static information used to typecheck and compile references to the unit's implementation. The unit's implementation must satisfy its interface. The unit's interface and the implementation may refer to antecedent units on which it depends.

The approach of distinguishing between the public interface and the private implementation of a unit has two useful properties. A unit may be implemented, typechecked and compiled as soon as the interfaces, but not necessarily any of the implementations, of its antecedents are available. Moreover, if a unit's implementation changes but its interface remains fixed, none of the units depending on it need to be re-typechecked or re-compiled.

The programming language Modula-2 [Wir88] is a good example of a language supporting this simple form of separate compilation. It also has a first-order module system. Modula-2 is particularly elegant because the notion of module coincides with the notion of compilation unit. In Modula-2, a module identifier is defined by giving both its private implementation and its public interface. Moreover, clients of a module are not allowed to assume any more about the module than is declared in its interface. In this way, it is possible to identify compilation unit interfaces with module interfaces, and compilation unit implementations with module implementations.

Ideally, as in Modula-2, in Standard ML the distinction between separate compilation and modularisation should merely be a matter of perspective:

we should be able to identify compilation units with modules. Like Modula-2, Standard ML's syntax supports a form of constrained module definition in which the implementation of the identifier is accompanied by an explicit signature. The implementation is constrained to match the signature. Unfortunately, as observed by Leroy [Ler94], the approach of identifying compilation units with Standard ML's constrained definitions does not succeed.

Motivated by the failure of this approach, Leroy [Ler94] proposes an alternative Modules calculus that, while preserving most of the flavour of Standard ML Modules, drops the distinction between syntactic and semantic objects, using signature expressions directly to type modules. Leroy's semantics provides good support for separate compilation.

Leroy's strategy for achieving separate compilation might lead one to believe that Standard ML's distinction between syntactic types and semantic objects cannot support separate compilation[1]. The purpose of the next two sections is to dispel this belief. We first analyse why the identification of compilation units with constrained module definitions fails. Instead of rejecting the semantics of Standard ML, we place the blame on an inappropriate choice of compilation unit. We then suggest an alternative notion of compilation unit that supports separate compilation.

Since we do not consider compilation issues in this thesis, we will only address the primary concern of separate typechecking of compilation units.

6.2 Identifying Compilation Units with Constrained Definitions

In Standard ML, as in Modula-2, it is possible to define a structure identifier by giving both its implementation and a signature expression used to constrain that implementation: the type of the implementation must match the signature. Similarly, one can constrain a functor definition by an explicit result signature: the type of the functor body must match this signature. In Mini-SML, the analog of Standard ML's syntax is obtained by adding the phrases:

$$\textbf{structure}\ \mathrm{X{:}S}\ =\ \mathrm{s;b}$$

$$\textbf{functor}\ \mathrm{F(X{:}S){:}S'}\ =\ \mathrm{s}\ \textbf{in}\ \mathrm{b}$$

to the syntax of structure bodies.

At first sight, this syntax seems to support separate compilation in the same way that the syntax of Modula-2 does. Since each module is declared

[1]We do not mean to imply that this is suggested by Leroy.

with a signature that its definition must match, it is tempting to identify a constrained definition with a compilation unit, treating the module identifier as the unit's name, the structure expression as its implementation and the signature as its interface. Intuitively, any program that is written as a sequence of constrained definitions should be "separately compilable".

Unfortunately, this approach fails: it is possible to write a "separately compilable" program that typechecks as a monolithic program, but contains a compilation unit that fails to typecheck when relying solely on the information provided by its antecedents' interfaces.

The reason this approach fails is quite simple. In Modula-2, a module's declared interface completely determines its typing properties — this is why it can be used as the interface of the corresponding compilation unit. In Standard ML, on the other hand, the signature of a constrained definition need not fully determine the typing properties of the defined module identifier. It is true that the module's implementation must match the signature; but the signature alone does not determine the identifier's interface to the rest of the program. This is because the explicit signature is merely used to *curtail* the type of its implementation. Consequently, the actual realisation of any type component that is merely specified but not defined in the signature is apparent to the rest of the program. If the correct classification of the remaining program depends on this realisation, then relying on the information in the signature alone can cause separate typechecking to fail. Moreover, if we replace the implementation by another, then the proviso that the replacement matches the same signature does not guarantee that the remaining program will continue to typecheck, because it may match the signature via a different realisation.

In Mini-SML terms, the semantics of Standard ML's constrained definitions are equivalent to the semantics derived from the following abbreviations:

$$\begin{aligned}\textbf{structure}\ \mathrm{X{:}S} \ =\ \mathrm{s;b} &\ \stackrel{\mathrm{def}}{=}\ \textbf{structure}\ \mathrm{X}\ =\ \mathrm{s} \succeq \mathrm{S;b}\\ \textbf{functor}\ \mathrm{F(X{:}S){:}S'}\ =\ \mathrm{s}\ \textbf{in}\ \mathrm{b} &\ \stackrel{\mathrm{def}}{=}\ \textbf{functor}\ \mathrm{F\ (X : S)}\ =\ \mathrm{s} \succeq \mathrm{S'}\ \textbf{in}\ \mathrm{b}\end{aligned}$$

Note that, in the expansion of each abbreviation, the constraining signature *curtails* its implementation.

It is easy to see why it is a mistake to identify compilation units with constrained definitions by examining the semantics of constrained definitions. Ignoring the side conditions on variable capture, the derived rule for the constrained structure definition is:

$$\frac{\begin{array}{l}\mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S} \\ \mathcal{C} \vdash \mathrm{S} \rhd \Lambda P'.\mathcal{S}' \\ \mathcal{S} \succeq \varphi\,(\mathcal{S}') \\ \mathrm{Dom}(\varphi) = P' \\ \mathcal{C}[\mathrm{X} : \varphi\,(\mathcal{S}')] \vdash \mathrm{b} : \exists P''.\mathcal{S}'' \\ \mathrm{X} \notin \mathrm{Dom}(\mathcal{S}'')\end{array}}{\mathcal{C} \vdash \textbf{structure}\ \mathrm{X{:}S}\ =\ \mathrm{s;b} : \exists P \cup P''.(\mathrm{X} : \varphi\,(\mathcal{S}'), \mathcal{S}'')}$$

Observe that the type $\varphi\,(\mathcal{S}')$ of the structure identifier X is determined both from the denotation $\Lambda P'.\mathcal{S}'$ of the signature S and from the type $\exists P.\mathcal{S}$ of its actual implementation s. This is because $\varphi\,(\mathcal{S}')$ incorporates the matching realisation φ. By preserving this realisation, the type of X contains more information than the signature alone. Moreover, it is this more informative type that serves as X's "interface" to the remaining definitions in b (notice that b is classified in the context $\mathcal{C}[\mathrm{X} : \varphi\,(\mathcal{S}')]$). Clearly, the static semantics of this phrase means that the signature, on its own, does not provide adequate information for typechecking b. For this reason, it is a mistake to consider S as the interface of X, and it should come as no surprise that the naive identification of compilation units with constrained definitions fails. Similar comments apply to the derived rule for a constrained functor definition.

6.3 Identifying Compilation Units with Abstracted Definitions

The previous discussion shows that the naive identification of compilation units with constrained definitions fails. But this does not imply that Standard ML is incompatible with separate compilation, as long as we can identify a better notion of compilation unit.

Contrast the curtailment rule:

$$\frac{\begin{array}{ll}\mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S} & \\ \mathcal{C} \vdash \mathrm{S} \rhd \Lambda P'.\mathcal{S}' & P \cap \mathrm{FV}(\Lambda P'.\mathcal{S}') = \emptyset \\ \mathcal{S} \succeq \varphi\,(\mathcal{S}') & \mathrm{Dom}(\varphi) = P'\end{array}}{\mathcal{C} \vdash \mathrm{s} \succeq \mathrm{S} : \exists P.\varphi\,(\mathcal{S}')}$$

that underlies the semantics of constrained definitions, with the abstraction rule:

$$\frac{\begin{array}{ll}\mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S} & \\ \mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P'.\mathcal{S}' & P \cap \mathrm{FV}(\Lambda P'.\mathcal{S}') = \emptyset \\ \mathcal{S} \succeq \varphi\,(\mathcal{S}') & \mathrm{Dom}(\varphi) = P'\end{array}}{\mathcal{C} \vdash \mathrm{s} \setminus \mathrm{S} : \exists P'.\mathcal{S}'}$$

While the type of the curtailment s $\succeq$ S depends on both the type of s and the denotation of S, the type $\exists P'.\mathcal{S}'$ of the abstraction s $\setminus$ S is fully determined solely by the signature's denotation $\Lambda P'.\mathcal{S}'$.

This observation suggests that abstractions may be used to enforce the requirement needed for separate compilation, namely, that a module's complete typing properties are captured by an explicit signature.

If we define *abstracted definitions* to be definitions of the form:

$$\textbf{structure}\ \mathrm{X}\ =\ \mathrm{s} \setminus \mathrm{S;b},$$

$$\textbf{functor}\ \mathrm{F}\ (\mathrm{X} : \mathrm{S})\ =\ \mathrm{s} \setminus \mathrm{S}'\ \textbf{in}\ \mathrm{b},$$

then the semantics of abstraction ensures that, in each case, the type of the module identifier is determined by its accompanying signature(s) alone. We claim that typechecking of the remaining definitions in b can proceed independently of the task of verifying that the implementation of the identifier is well-typed and matches its signature.

We can justify this claim by examining the derived rules for abstracted definitions. For instance, ignoring the side conditions to prevent variable capture, the derived rule for an abstracted structure definition is:

$$\frac{\begin{array}{l}\mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S} \\ \mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P'.\mathcal{S}' \\ \mathcal{S} \succeq \varphi\,(\mathcal{S}') \\ \mathrm{Dom}(\varphi) = P' \\ \mathcal{C}[\mathrm{X} : \mathcal{S}'] \vdash \mathrm{b} : \exists P''.\mathcal{S}'' \\ \mathrm{X} \notin \mathrm{Dom}(\mathcal{S}'')\end{array}}{\mathcal{C} \vdash \textbf{structure}\ \mathrm{X}\ =\ \mathrm{s} \setminus \mathrm{S;b} : \exists P' \cup P''.(\mathrm{X} : \mathcal{S}', \mathcal{S}'')}$$

Observe that, provided the signature denotes, then the classification of the entire phrase can be split into two *independent* subtasks. The first task corresponds to verifying that the implementation s matches the signature (checking $\mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S}$ and $\mathcal{S} \succeq \varphi\,(\mathcal{S}')$ with Dom(φ)=P'). The second task corresponds to classifying the remaining definitions of b (checking $\mathcal{C}[\mathrm{X} : \mathcal{S}'] \vdash \mathrm{b} : \exists P''.\mathcal{S}''$ and $\mathrm{X} \notin \mathrm{Dom}(\mathcal{S}'')$). The tasks are independent, because the parameters required to carry out each task, namely P' and $\mathcal{S}'$, are determined by the signature alone. A similar division into independent subtasks arises from the derived rule for an abstracted functor definition.

We thus obtain a simple solution to the separate compilation problem: identify compilation units with *abstracted definitions* and define a *separately compilable* program to be a program that is written as a sequence of abstracted definitions. For a program written in this style, each unit's implementation can be typechecked by relying solely on the signature(s) of its antecedent units. Moreover, the type (and typability) of the program is the same, irrespective of whether we choose to check it as a whole or to check its units separately.

Remark 6.3.1 *(Compilation Units for Higher-Order Modules).* Although our discussion has focussed on first-order Modules, the proposal to identify compilation units with abstracted definitions applies equally well to Higher-Order Modules, and for the same reasons. Indeed, it is even easier in the higher-order case because we already have a notion of functor interface: the functor signature. Moreover, since structure and functor definitions are subsumed by a single notion of module definition, we can get away with a uniform treatment of both by identifying compilation units with *abstracted module definitions*:

$$\textbf{module}\ \mathrm{X} = \mathrm{m} \setminus \mathrm{S}; \mathrm{b}.$$

Remark 6.3.2. Strictly speaking, Leroy's criticism that the original version of Standard ML [MTH90] does not support separate compilation is valid, but only because of syntactic deficiencies of the language. This version does not support abstractions, although they are mentioned in MacQueen's original design [HMM86] and their adoption is discussed in the Commentary [MT91]. Furthermore, even with their adoption, the systematic use of abstractions to achieve separate compilation is too restrictive for practical programming. This is due to shortcomings in the syntax of signature expressions. In this version of Standard ML, signatures cannot specify concrete type definitions. This has the unfortunate consequence of forcing the programmer to hide the definitions of all type components of an abstracted structure. Type sharing constraints alleviate this restriction somewhat but not in a fully general manner. These problems disappear in the revision of Standard ML [MTH96]. It supports abstractions and the syntax of signature expressions has been changed to include type definitions.

6.4 A Lingering Problem: The Lack of Syntactic Interfaces

In the previous section, we identified a programming style that supports separate compilation. As long as the programmer adheres to this style, she can separately type-check (and compile) the components of her program. The style is flexible enough for practical programming, because the syntax of signatures provides fine control over the abstraction of individual type components.

There remains, however, a sense in which this approach is unsatisfactory and offers only a partial solution. As we will soon see, in Mini-SML (and, by extension, Standard ML), it is not always possible to fully specify the type of a structure expression using a signature. Consequently, it is possible to write a program consisting of a sequence of ordinary module definitions that typechecks, but which cannot be re-written as a sequence of compilation units by surrounding each module's implementation with an abstraction. In theory, this can prevent a programmer from decomposing a monolithic program into a sequence of separate compilation units that still typecheck.

From a practical, software engineering perspective, this is a minor flaw: the primary motivation for identifying a notion of compilation unit is to support the incremental *construction* of programs (not the inverse process). Nevertheless, it is worth pointing out why the problem arises. In the remainder of this chapter, we illustrate the problem and propose modifications to the semantics that make it disappear. The main point of this work is not to suggest that these modifications are urgently required, but merely to clarify exactly how far the semantics is from enjoying the property that every module expression admits a syntactic representation of its type.

6.4.1 Eclipsed Identifiers

In Mini-SML, as in Standard ML, it is possible to redeclare an identifier which is already declared in the current context. This is useful, as it allows the same component name to be re-used within substructures and subsignatures. The meaning of an identifier is resolved by static scoping: each occurrence of an identifier refers to its textually most recent declaration.

Unfortunately, this simple scheme has a flaw: it prevents a programmer from referring to two distinct declarations of the same identifier in situations where it is necessary to do so: any earlier declaration is *eclipsed* by the shadow of the most recent declaration. In turn, this can prevent the programmer from expressing the full type of a module using a signature.

```
structure X = (struct type t = int end
                \ sig type t : 0 end);
structure Y = (struct local A  =  X in
                        structure X = (struct type u = bool end
                                        \ sig type u : 0 end);
                        type v = A.t → X.u
               end
               \ sig structure X : sig type u : 0 end;
                      type v = $\boxed{?}$ → X.u
               end)
```

Figure 6.1: A structure with an inexpressible type because of an eclipsed identifier.

```
structure X = (struct type t = int end
                \ sig type t : 0 end);
functor F (A : sig type t : 0 end) =
          struct structure X = (struct type u = bool end
                                 \ sig type u : 0 end);
                 type v = A.t → X.u
          end
in
structure Y = (F X)  \ (sig structure X : sig type u : 0 end;
                             type v = $\boxed{?}$ → X.u
                         end)
```

Figure 6.2: Another structure with an inexpressible type because of an eclipsed identifier.

Example 6.4.1. To see why, consider the (contrived) example in Figure 6.1. The structure expression in the definition of **Y** has a type, but it is impossible to give a signature expression that fully specifies this type. To specify **Y**'s type we need to be able to complete the specification of its type component **v** by filling in the $\boxed{?}$. Unfortunately, we are prevented from doing this because the domain and range of **v**'s definition are types that cannot be specified in a common context. The domain of **v** can only be specified as **X**.**t** in a context where the outermost definition of **X** is in scope. The range of **v** can only be specified as **X**.**u** in a context where the inner specification of **X** is in scope. Since the definition and specification of **X** both declare the *same* identifier, one must eclipse the other and they cannot be in scope at the same time. Because the actual definition of its **v**-component cannot be specified, the type of **Y** cannot be fully captured by a signature.

The example in Figure 6.2, that uses a functor definition instead of a local definition, demonstrates the same flaw, indicating that the cause does not lie with allowing local definitions.

This problem with eclipsed identifiers has been noted before by Harper and Lillibridge [HL94]. They suggest a solution to this problem that relies on distinguishing between external and internal component identifiers. This solution is satisfactory, but it requires the programmer to maintain two name-spaces, which can be inconvenient.

We will sketch another solution, that relies on distinguishing declarations by their binding depth. Observe that the example above is problematic only because the syntax of Mini-SML does not allow us to distinguish between different declarations of the same identifier, in this case the structure identifier **X**. Every reference to an identifier is resolved by static scoping. In the semantics, this behaviour is ensured by defining contexts as finite maps. Extending a context by a new declaration overrides any previous declaration of that identifier.

Our alternative solution relies on defining contexts, not as finite-maps, but as *lists* of declarations. New declarations are added to the head of the list, without forgetting the effect of previous declarations. In this way, all declarations are preserved in the inverse order in which they were added.

To provide access to all declarations in the context, each reference to an identifier must now be accompanied by an index indicating the depth of the intended declaration. The depth is understood to be relative to the depth of other declarations of the same identifier, counting from the head of the context. Intuitively, the reference $\mathbf{i}^n$, where $\mathbf{i}$ is an identifier and $n \geq 0$ is an index, references the n-th-most recent declaration of $\mathbf{i}$ in the current context.

```
sig structure X : sig type u : 0 end;
    type v = X^1.t → X.u
end
```

Figure 6.3: The signature of **Y** using indexed identifiers.

For convenience, we adopt the convention that the reference **i**, lacking an index, is an abbreviation for $\mathbf{i}^0$, i.e. the most recent declaration of **i**. We will formalise this mechanism in Section 6.5.

Example 6.4.2. The signature in Figure 6.3 exploits an indexed reference to fully specify the problematic type of the structure expression **Y** in Figure 6.1.

Our technique is essentially a combination of named identifiers and de Bruijn [deB72] indices. Although terms written in pure de Bruijn notation are notoriously difficult for humans to read, our scheme seems more acceptable in realistic programming situations. First, we need only use an index when we need to refer to an eclipsed identifier (this rarely occurs in practice and can easily be avoided by disciplined programming). Second, the counting scheme is relative to identifiers of the same name: hence indices, when they need to be used, are small and manageable.

6.4.2 Anonymous Abstract Types

There is another, more problematic phenomenon that can prevent a structure expression s from admitting a complete specification of its type as a signature S. It is possible for the type $\exists P.\mathcal{S}$ of the structure expression to contain an existentially quantified variable $\alpha \in P$, i.e. an abstract type, that is *anonymous*, in the sense that α does not occur as the denotation of a type component within the semantic object $\mathcal{S}$. For the signature S to completely specify the type of the structure s, there must be a one-to-one correspondence between the parameters P' of the signature's denotation $\Lambda P'.\mathcal{S}'$ and the abstract types P that are existentially quantified in the structure's type $\exists P.\mathcal{S}$. The problem is that a signature parameter can never be anonymous. So if the type $\exists P.\mathcal{S}$ quantifies an anonymous abstract type, no such correspondence can exist.

Example 6.4.3. Consider the definition of the structure **X** in Figure 6.4. It is well-typed, but its type contains an anonymous abstract type.

```
structure X =
      ((struct type t = int;
                type u = int → int
        end
        \ sig type t : 0;
              type u = int → t
          end)
        ⪰ sig type u : 0
          end)
```

Figure 6.4: A structure with an anonymous abstract type.

```
functor F (A : sig type u : 0 end)  =  A
in
structure X = F  (struct type t = int;
                         type u = int → int
                  end
                  \ sig type t : 0;
                        type u = int → t
                    end)
```

Figure 6.5: Another structure with an anonymous abstract type.

The type of the innermost structure expression is:

$$\exists\emptyset.(\mathbf{t} = \mathbf{int}, \mathbf{u} = \mathbf{int} \to \mathbf{int}).$$

The type of the inner abstraction that hides the implementation of all occurrences of **t** is:

$$\exists\{\alpha\}.(\mathbf{t} = \alpha, \mathbf{u} = \mathbf{int} \to \alpha).$$

Finally, the type of the curtailment that forgets the type component **t**, yet preserves the implementation of **u**, is:

$$\exists\{\alpha\}.(\mathbf{u} = \mathbf{int} \to \alpha).$$

Notice that the abstract type α is anonymous.

Unfortunately, there is no signature expression that, when used as an abstraction, fully specifies the type of **X**'s implementation. To see why, suppose, to the contrary, that S is such a signature expression. Inverting the abstraction rule, it must be that case that S denotes the semantic signature:

$$\Lambda\{\alpha\}.(\mathbf{u} = \mathbf{int} \to \alpha).$$

This signature is not solvable in the sense of Definition 4.2 (Chapter 4), precisely because the type parameter α is anonymous in $(\mathbf{u} = \mathbf{int} \to \alpha)$. This contradicts Lemma 4.3 (Solvability), that states that the denotation of a signature must be solvable. It follows that S cannot exist.

The example in Figure 6.5, that uses functor application instead of curtailment, demonstrates the same flaw, indicating that anonymous abstract types do not only arise from curtailment phrases.

This raises a natural question: can we modify the semantics in such a way that every module expression admits a signature specifying its type? The observation that the existential quantification in module types cannot be captured by the restricted form of type parameterisation afforded by signature expressions suggests two ways to proceed.

The first is to generalise signature expressions to provide a finer degree of control of type parameterisation: unfortunately, it is not clear how to do this in a manner that preserves the fundamental properties ensured by solvable signatures: the decidability and existence of unique solutions to signature matching problems.

A more radical proposal is to abandon the use of existential quantification altogether. Observe that the only construct that introduces existential types is the abstraction s \ S. Deleting the phrase from the language allows us to simplify the static semantics dramatically. Structure expressions can be classified by simple semantic structures (as opposed to existential structures). Moreover, the classification rules no longer have to implicitly eliminate and re-introduce existential quantifiers when determining the classification of a phrase from the classification of its subphrases.

The loss of the abstraction phrase is lamentable. Without it, there is no way to isolate a structure from its program context. In Section 5.2.2, we discussed how abstractions permit the programmer to change the realisation of type components within a structure, without this change affecting the typability of the surrounding program.

Fortunately, it is possible to formulate a weaker form of abstraction without resorting to existential quantification. Intuitively, the idea is to strike a compromise between the semantics of constrained and abstracted module definitions. We retain the syntax of constrained definitions but suggest a different semantics:

$$\frac{\begin{array}{l}\mathcal{C} \vdash \mathrm{s} : \mathcal{S} \\ \mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{S}' \\ \mathcal{S} \succeq \varphi\,(\mathcal{S}') \\ \mathrm{Dom}(\varphi) = P \\ P \cap \mathrm{FV}(\mathcal{C}) = \emptyset \\ \mathcal{C}[\mathrm{X} : \mathcal{S}'] \vdash \mathrm{b} : \mathcal{S}'' \\ \mathrm{X} \notin \mathrm{Dom}(\mathcal{S}'')\end{array}}{\mathcal{C} \vdash \textbf{structure}\ \mathrm{X{:}S} \ = \ \mathrm{s;b} : \mathrm{X} : \varphi\,(\mathcal{S}'), \varphi\,(\mathcal{S}'')}$$

The novelty of this rule lies in the treatment of the signature constraint. The variables P of the signature are treated as parameters during the classification of b. Thus their actual realisation, φ, cannot affecting the typability of b. This allows X's actual implementation s to change, provided it continues to match the signature. The realisation is not abstracted, however. Instead, it is *applied* to the result type $\mathrm{X} : \varphi\,(\mathcal{S}'), \varphi\,(\mathcal{S}'')$. This final step of discharging the parameters manages to avoid the introduction of existential quantification over P.

Notice that the signature S fully determines X's interface to the remaining definitions in b, providing good support for separate compilation. However, any change in realisation will be reflected in the type of the complete phrase, even though it cannot affect the *typability* of the remaining definitions. In short, the realisation of the module is locally abstract, but globally transparent. Indeed, the semantics of the phrase **module** X : S = s;b is very similar to the semantics of the functor application

$$(\textbf{functor}(\mathrm{X}' : \mathrm{S})\textbf{struct module}\ \mathrm{X} = \mathrm{X}'; \mathrm{b}\ \textbf{end})\ \mathrm{s},$$

except that the former is a structure body, while the latter is structure expression.

The advantage of this semantics is the following. If such a definition occurs at the top-level of a program, then it may be treated as a compilation unit, without relying on existential types. The disadvantage of this semantics is that if the definition occurs, not at the top-level, but deeper within the program, then any change in its realisation may affect the program's typability. In summary, the phrase provides a weak form of abstraction that is adequate for separate compilation, but cannot replace the role of arbitrary abstractions.

6.5 A Module Language with Syntactic Representations of Types

The aim of this section is to sketch a modules language that, by adopting the changes discussed in Sections 6.4.1 and 6.4.2, namely the introduction of indexed identifiers to prevent eclipsing, and the removal of abstractions to rule out anonymous abstract types, enjoys the following, informal property:

Property 6.1 (Representation). *Every well-typed module expression admits a signature that fully specifies its type.*

Since Higher-Order Modules is already equipped with a notion of functor signature, and has a uniform treatment of structures and functors, it is easier to design our language as a variant of Higher-Order Modules. However, for the results in this section, the sheer size of a full Modules and Core language begins to get in the way of feasible pencil and paper proof. To simplify the argument, we will eliminate the Core by considering a skeletal Modules language with a fixed grammar of definable types and no value definitions or specifications. A further simplification is to restrict our attention to a language with a single kind ($\star$) of (non-parameterised) definable types.

Figure 6.6 defines the grammar of our language. Most of its phrases should be familiar from the full definition of Higher-Order Modules. Note that type and module declarations are referenced by indexed identifiers, as motivated in Section 6.4.1. The abstraction phrase $\mathrm{m} \setminus \mathrm{S}$ has been removed in favour of the constrained definition **module** $\mathrm{X} : \mathrm{S} \ = \ \mathrm{m;b}$. This is the higher-order version of the phrase proposed in Section 6.4.2.

The semantic objects of our language are defined in Figure 6.7. They are derived from the semantic objects of Higher-Order Modules, with the following modifications. The set of Core kinds has been replaced by a single

TypId	$\overset{\text{def}}{=}$	$\{\mathbf{t}, \mathbf{u}, \ldots\}$	type identifiers
ModId	$\overset{\text{def}}{=}$	$\{\mathbf{X}, \mathbf{Y}, \mathbf{F}, \mathbf{G}, \ldots\}$	module identifiers
d	::=	t^n	*indexed type identifier*
	|	m.t	type projection
	|	$\mathrm{d} \to \mathrm{d}$	function space
B	::=	**type** t = d; B	type definition
	|	**type** t;B	type specification
	|	**module** X : S; B	module specification
	|	ϵ_{B}	empty body
S	::=	**sig** B **end**	structure signature
	|	**funsig**(X:S)S′	functor signature
b	::=	**type** t = d; b	type definition
	|	**module** X = m; b	module definition
	|	**module** X : S = m;b	*constrained module definition*
	|	**local** X = m **in** b	local module definition
	|	ϵ_{b}	empty body
m	::=	X^n	*indexed module identifier*
	|	m.X	submodule projection
	|	**struct** b **end**	structure
	|	**functor**(X : S)m	functor
	|	m m′	functor application
	|	$\mathrm{m} \succeq \mathrm{S}$	signature curtailment

Figure 6.6: Grammar

$\kappa \in \mathit{Kind}$	$::=$	$\star$	*base kind*
	$\mid$	$\kappa \to \kappa'$	function space
$\alpha^\kappa \in \mathit{TypVar}^\kappa$	$\stackrel{\text{def}}{=}$	$\{\alpha^\kappa, \beta^\kappa, \delta^\kappa, \gamma^\kappa, \ldots\}$	type variables
$\alpha \in \mathit{TypVar}$	$\stackrel{\text{def}}{=}$	$\biguplus_{\kappa \in \mathit{Kind}} \mathit{TypVar}^\kappa$	
$P, Q, \ldots \in \mathit{TypVarSet}$	$\stackrel{\text{def}}{=}$	$\mathrm{Fin}(\mathit{TypVar})$	type variable sets
$d \in \mathit{DefTyp}$	$::=$	$\nu^\star$	type name
	$\mid$	$d \to d'$	function space
$\nu^\kappa \in \mathit{TypNam}^\kappa$	$::=$	α^κ	type variable
	$\mid$	$\nu^{\kappa' \to \kappa}\ \tau^{\kappa'}$	application
$\tau^\kappa \in \mathit{Typ}^\kappa$	$::=$	d (provided $\kappa \equiv \star$)	definable type
	$\mid$	$\Lambda\alpha^{\kappa'}.\tau^{\kappa''}$ (provided $\kappa \equiv \kappa' \to \kappa''$)	type abstraction
	$\mid$	ν^κ	type name
$\tau \in \mathit{Typ}$	$\stackrel{\text{def}}{=}$	$\biguplus_{\kappa \in \mathit{Kind}} \mathit{Typ}^\kappa$	
$\mathcal{S} \in \mathit{Str}$	$::=$	$\mathrm{t} = \tau^\star, \mathcal{S}$ (provided $\mathrm{t} \notin \mathrm{Dom}(\mathcal{S})$)	type component
	$\mid$	$\mathrm{X} : \mathcal{M}, \mathcal{S}$ (provided $\mathrm{X} \notin \mathrm{Dom}(\mathcal{S})$)	module component
	$\mid$	$\epsilon_\mathcal{S}$	empty structure
$\mathcal{F} \in \mathit{Fun}$	$::=$	$\forall P.\mathcal{M} \to \mathcal{M}'$	functor
$\mathcal{M} \in \mathit{Mod}$	$::=$	$\mathcal{S}$	structure
	$\mid$	$\mathcal{F}$	functor
$\mathcal{L} \in \mathit{Sig}$	$::=$	$\Lambda P.\mathcal{M}$	signature
$\mathcal{C} \in \mathit{Context}$	$::=$	$\mathcal{C}[\mathrm{t} = \tau^\star]$	*type declaration*
	$\mid$	$\mathcal{C}[\mathrm{X} : \mathcal{M}]$	*module declaration*
	$\mid$	$\epsilon_\mathcal{C}$	*empty context*

Figure 6.7: Semantic Objects

base kind $\star$ classifying definable types. Semantic structures and contexts no longer declare value components. The removal of the abstraction phrase means that existentially quantified types are no longer needed. Finally, instead of being finite maps, contexts are defined inductively as *lists* of declarations to support the use of indexed identifiers. For readability, contexts are defined to extend to the right[2], so that the *head* of a context is its rightmost declaration, and its *tail* the context preceding that declaration.

The definition of the enrichment relations, $\_ \succeq \_$, between structures, functors and modules is as for Higher-Order Modules (Definition 5.14), except that the premise concerning value components is deleted from the rule relating structures (Rule ($\succeq$ -1)).

The definition of the operation $\hat{\eta}(\_)$, converting a type of kind $\star$ into an equivalent definable type, degenerates to:

$$\begin{array}{rcl} \hat{\eta}(\_) & \in & \mathit{Typ}^{\star} \rightarrow \mathit{DefTyp} \\ \hat{\eta}(\nu) & \overset{\text{def}}{=} & \nu \\ \hat{\eta}(d) & \overset{\text{def}}{=} & d\,. \end{array}$$

We can now present the judgements and rules of the static semantics. The rules defining the denotation of definable types are straightforward. The only novelty lies in Rules (1)-(4) that formalise the retrieval of the nth-most recent declaration of a type identifier. The denotation judgements for signature expressions and bodies are essentially unchanged. The classification rules for structure bodies and expressions are degenerate instances of the corresponding rules in Higher-Order Modules. Since the abstraction phrase has been removed, we can classify module expression by module types $\mathcal{M} \in \mathit{Mod}$ instead of existential module types $\exists P.\mathcal{M}$. This simplifies the rules considerably because they no longer need to perform any implicit elimination and introduction of existential quantifiers. Rules (18)-(21) formalise the retrieval of the nth-most recent declaration of a module identifier; their definition is analogous to Rules (1)-(4).

Definable Types $\boxed{\mathcal{C} \vdash \mathrm{d} \triangleright d}$

$$\frac{}{\mathcal{C}[\mathrm{t} = \tau] \vdash \mathrm{t}^{0} \triangleright \hat{\eta}(\tau)} \tag{1}$$

$$\frac{\mathcal{C} \vdash \mathrm{t}^{n} \triangleright d}{\mathcal{C}[\mathrm{t} = \tau] \vdash \mathrm{t}^{n+1} \triangleright d} \tag{2}$$

[2]This goes against the usual convention that lists extend to the *left*, but this is just a matter of syntax.

$$\frac{\mathcal{C} \vdash \mathrm{t}^n \triangleright d \quad \mathrm{t} \neq \mathrm{t}'}{\mathcal{C}[\mathrm{t}' = \tau] \vdash \mathrm{t}^n \triangleright d} \tag{3}$$

$$\frac{\mathcal{C} \vdash \mathrm{t}^n \triangleright d}{\mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash \mathrm{t}^n \triangleright d} \tag{4}$$

$$\frac{\mathcal{C} \vdash \mathrm{m} : \mathcal{S} \quad \mathrm{t} \in \mathrm{Dom}(\mathcal{S}) \quad \mathcal{S}(\mathrm{t}) = \tau}{\mathcal{C} \vdash \mathrm{m.t} \triangleright \hat{\eta}(\tau)} \tag{5}$$

$$\frac{\mathcal{C} \vdash \mathrm{d} \triangleright d \quad \mathcal{C} \vdash \mathrm{d}' \triangleright d'}{\mathcal{C} \vdash \mathrm{d} \to \mathrm{d}' \triangleright d \to d'} \tag{6}$$

Signature Bodies $\boxed{\mathcal{C} \vdash \mathrm{B} \triangleright \mathcal{L}}$

$$\frac{\begin{array}{ll} \mathcal{C} \vdash \mathrm{d} \triangleright d & P \cap \mathrm{FV}(d) = \emptyset \\ \mathcal{C}[\mathrm{t} = d] \vdash \mathrm{B} \triangleright \Lambda P.\mathcal{S} & \mathrm{t} \notin \mathrm{Dom}(\mathcal{S}) \end{array}}{\mathcal{C} \vdash \textbf{type}\ \mathrm{t} = \mathrm{d}; \mathrm{B} \triangleright \Lambda P.\mathrm{t} = d, \mathcal{S}} \tag{7}$$

$$\frac{\mathcal{C}[\mathrm{t} = \alpha^\star] \vdash \mathrm{B} \triangleright \Lambda P.\mathcal{S} \quad \alpha^\star \notin \mathrm{FV}(\mathcal{C}) \cup P \quad \mathrm{t} \notin \mathrm{Dom}(\mathcal{S})}{\mathcal{C} \vdash \textbf{type}\ \mathrm{t;B} \triangleright \Lambda\{\alpha^\star\} \cup P.\mathrm{t} = \alpha^\star, \mathcal{S}} \tag{8}$$

$$\frac{\begin{array}{ll} \mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{M} & \\ \mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash \mathrm{B} \triangleright \Lambda Q.\mathcal{S} & P \cap \mathrm{FV}(\mathcal{C}) = \emptyset \\ Q \cap (P \cup \mathrm{FV}(\mathcal{M})) = \emptyset & \mathrm{X} \notin \mathrm{Dom}(\mathcal{S}) \end{array}}{\mathcal{C} \vdash \textbf{module}\ \mathrm{X} : \mathrm{S}; \mathrm{B} \triangleright \Lambda P \cup Q.\mathrm{X} : \mathcal{M}, \mathcal{S}} \tag{9}$$

$$\frac{}{\mathcal{C} \vdash \epsilon_{\mathrm{B}} \triangleright \Lambda\emptyset.\epsilon_{\mathcal{S}}} \tag{10}$$

Signature Expressions $\boxed{\mathcal{C} \vdash \mathrm{S} \triangleright \mathcal{L}}$

$$\frac{\mathcal{C} \vdash \mathrm{B} \triangleright \mathcal{L}}{\mathcal{C} \vdash \textbf{sig}\ \mathrm{B}\ \textbf{end} \triangleright \mathcal{L}} \tag{11}$$

$$\frac{\begin{array}{l}\mathcal{C} \vdash \mathrm{S} \rhd \Lambda P.\mathcal{M} \\ P \cap \mathrm{FV}(\mathcal{C}) = \emptyset \quad P = \{\alpha_0^{\kappa_0}, \ldots, \alpha_{n-1}^{\kappa_{n-1}}\} \\ \mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash \mathrm{S}' \rhd \Lambda Q.\mathcal{M}' \\ Q' \cap (P \cup \mathrm{FV}(\mathcal{M}) \cup \mathrm{FV}(\Lambda Q.\mathcal{M}')) = \emptyset \\ [Q'/Q] = \{\beta^\kappa \mapsto \beta^{\kappa_0 \to \cdots \kappa_{n-1} \to \kappa}\ \alpha_0 \cdots \alpha_{n-1} | \beta^\kappa \in Q\} \\ Q' = \{\beta^{\kappa_0 \to \cdots \kappa_{n-1} \to \kappa} | \beta^\kappa \in Q\}\end{array}}{\mathcal{C} \vdash \mathbf{funsig}(\mathrm{X}{:}\mathrm{S})\mathrm{S}' \rhd \Lambda Q'.\forall P.\mathcal{M} \to [Q'/Q]\,(\mathcal{M}')} \quad (12)$$

Structure Bodies $\boxed{\mathcal{C} \vdash \mathrm{b} : \mathcal{S}}$

$$\frac{\mathcal{C} \vdash \mathrm{d} \rhd d \quad \mathcal{C}[\mathrm{t} = d] \vdash \mathrm{b} : \mathcal{S} \quad \mathrm{t} \notin \mathrm{Dom}(\mathcal{S})}{\mathcal{C} \vdash \mathbf{type}\ \mathrm{t} = \mathrm{d}; \mathrm{b} : \mathrm{t} = d, \mathcal{S}} \quad (13)$$

$$\frac{\mathcal{C} \vdash \mathrm{m} : \mathcal{M} \quad \mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash \mathrm{b} : \mathcal{S} \quad \mathrm{X} \notin \mathrm{Dom}(\mathcal{S})}{\mathcal{C} \vdash \mathbf{module}\ \mathrm{X} = \mathrm{m}; \mathrm{b} : \mathrm{X} : \mathcal{M}, \mathcal{S}} \quad (14)$$

$$\frac{\begin{array}{ll}\mathcal{C} \vdash \mathrm{m} : \mathcal{M} & \\ \mathcal{C} \vdash \mathrm{S} \rhd \Lambda P.\mathcal{M}' & \mathcal{M} \succeq \varphi\,(\mathcal{M}') \\ \mathrm{Dom}(\varphi) = P & P \cap \mathrm{FV}(\mathcal{C}) = \emptyset \\ \mathcal{C}[\mathrm{X} : \mathcal{M}'] \vdash \mathrm{b} : \mathcal{S} & \mathrm{X} \notin \mathrm{Dom}(\mathcal{S})\end{array}}{\mathcal{C} \vdash \mathbf{module}\ \mathrm{X} : \mathrm{S}\ =\ \mathrm{m};\mathrm{b} : (\mathrm{X} : \varphi\,(\mathcal{M}'), \varphi\,(\mathcal{S}))} \quad (15)$$

$$\frac{\mathcal{C} \vdash \mathrm{m} : \mathcal{M} \quad \mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash \mathrm{b} : \mathcal{S}}{\mathcal{C} \vdash \mathbf{local}\ \mathrm{X}\ =\ \mathrm{m}\ \mathbf{in}\ \mathrm{b} : \mathcal{S}} \quad (16)$$

$$\frac{}{\mathcal{C} \vdash \epsilon_{\mathrm{b}} : \epsilon_{\mathcal{S}}} \quad (17)$$

Module Expressions $\boxed{\mathcal{C} \vdash \mathrm{m} : \mathcal{M}}$

$$\frac{}{\mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash \mathrm{X}^0 : \mathcal{M}} \quad (18)$$

$$\frac{\mathcal{C} \vdash \mathrm{X}^n : \mathcal{M}}{\mathcal{C}[\mathrm{X} : \mathcal{M}'] \vdash \mathrm{X}^{n+1} : \mathcal{M}} \quad (19)$$

$$\frac{\mathcal{C} \vdash \mathrm{X}^n : \mathcal{M} \quad \mathrm{X} \neq \mathrm{X}'}{\mathcal{C}[\mathrm{X}' : \mathcal{M}'] \vdash \mathrm{X}^n : \mathcal{M}} \quad (20)$$

$$\frac{\mathcal{C} \vdash \mathrm{X}^n : \mathcal{M}}{\mathcal{C}[\mathrm{t} = \tau] \vdash \mathrm{X}^n : \mathcal{M}} \tag{21}$$

$$\frac{\mathcal{C} \vdash \mathrm{m} : \mathcal{S} \quad \mathrm{X} \in \mathrm{Dom}(\mathcal{S}) \quad \mathcal{S}(\mathrm{X}) = \mathcal{M}}{\mathcal{C} \vdash \mathrm{m.X} : \mathcal{M}} \tag{22}$$

$$\frac{\mathcal{C} \vdash \mathrm{b} : \mathcal{M}}{\mathcal{C} \vdash \textbf{struct}\ \mathrm{b}\ \textbf{end} : \mathcal{M}} \tag{23}$$

$$\frac{\mathcal{C} \vdash \mathrm{S} \rhd \Lambda P.\mathcal{M} \quad P \cap \mathrm{FV}(\mathcal{C}) = \emptyset \quad \mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash \mathrm{m} : \mathcal{M}'}{\mathcal{C} \vdash \textbf{functor}(\mathrm{X} : \mathrm{S})\mathrm{m} : \forall P.\mathcal{M} \to \mathcal{M}'} \tag{24}$$

$$\frac{\mathcal{C} \vdash \mathrm{m} : \forall Q.\mathcal{M}' \to \mathcal{M} \quad \mathcal{C} \vdash \mathrm{m}' : \mathcal{M}'' \quad \mathcal{M}'' \succeq \varphi(\mathcal{M}') \quad \mathrm{Dom}(\varphi) = Q}{\mathcal{C} \vdash \mathrm{m}\ \mathrm{m}' : \varphi(\mathcal{M})} \tag{25}$$

$$\frac{\mathcal{C} \vdash \mathrm{m} : \mathcal{M}' \quad \mathcal{C} \vdash \mathrm{S} \rhd \Lambda P.\mathcal{M} \quad \mathcal{M}' \succeq \varphi(\mathcal{M}) \quad \mathrm{Dom}(\varphi) = P}{\mathcal{C} \vdash \mathrm{m} \succeq \mathrm{S} : \varphi(\mathcal{M})} \tag{26}$$

6.5.1 A Strategy to Establish the Representation Property

We can express the representation property more formally as the following requirement: if $\mathcal{C} \vdash \mathrm{m} : \mathcal{M}$ then there is some signature S such that $\mathcal{C} \vdash \mathrm{S} \rhd \Lambda\emptyset.\mathcal{M}$. Insisting on an empty set of parameters in the denotation of S ensures that S is a complete specification of the type of m. From now on, we shall refer to such a signature as a *representation* of $\mathcal{M}$.

Unfortunately, we cannot simply recover a representation by defining an inductive translation of the semantic object. Semantic objects arise by *erasing* the dependency of type phrases on module terms. Thus, given a arbitrary semantic object, it is typically impossible to infer the module terms that produced it.

Instead, we will need to use a more refined strategy that constructs a representation by induction on the *derivation* of $\mathcal{C} \vdash \mathrm{m} : \mathcal{M}$. The idea is to incrementally construct the representation of $\mathcal{M}$ from *both* the subphrases of m and the representations obtained for their types.

For example, in the case of a well-typed application m m′, if m's type has the representation **funsig**(**X**:S)S′, then the type of the application may be represented by the derived signature [m′/X]S′, obtained by *substituting* the actual argument in the functor signature's range. Of course, this means that we will need to define a notion of substitution on phrases.

In the case of a projection m.X, a representation can be obtained by *projecting* the corresponding subsignature from the representation of m's type. This requires care, because X's specification may depend on components specified earlier in the enclosing signature.

In the case of an identifier occurrence X^n, we will need to construct a representation from the signature used to introduce that identifier. To prove that the representation property holds, we will therefore need to maintain the invariant that every module identifier in the context admits a signature denoting its type. If X occurs in a phrase of the form **functor**(X : S)... X ... or **module** X : S = m;... X ... then the signature S introducing X into the context may not itself be a representation, because its denotation may contain a non-empty set of parameters. Fortunately, we can define an operation that *strengthens* the original signature to yield a representation of X's type in the context of its declaration.

6.5.2 Syntactic Operations on Phrases

In this section, we define the syntactic operations needed to establish the representation property.

The first operation we will need is a notion of substituting a module expression for a module identifier, and a definable type for a type identifier. This is the technical motivation for introducing de Bruijn indexed identifiers. The indices allow us to define substitution in a way that avoids the capture of free identifiers.

The definition of substitution relies on an auxiliary operation on phrases, called *lifting*. Its definition is derived from the standard definition of lifting used to implement substitution for pure de Bruijn terms. Lifting is a ternary operation, written $\mathrm{p} \upharpoonleft^n_{\mathrm{X}}$. Lifting modifies the phrase p to avoid the capture of any free occurrences of the module identifier X. It is used when substituting p into the scope of a new declaration of X. Intuitively, lifting increments the free occurrences of X within p to refer past one extra declaration of X. The additional argument n is just a counter of the number of declarations of X enclosing p. The counter keeps track of which occurrences of X are free, and which are bound. Thus, when we encounter the phrase X^i, if $i \geq n$, the occurrence is free and lifted to X^{i+1}; otherwise, the occurrence is bound by one of the n enclosing declarations of X and is left unchanged. The operation $\mathrm{p} \upharpoonleft^n_{\mathrm{t}}$ for lifting a type identifier is analogous.

Definition 6.2 (Lifting). We will only show a few cases of the definition. The others are similar.

Lifting of module identifiers is defined as:

$$\begin{array}{rcl}
\mathrm{X'}^i \uparrow^n_{\mathrm{X}} & \stackrel{\text{def}}{=} & \begin{cases} \mathrm{X}^{i+1} & \text{if } \mathrm{X} = \mathrm{X'} \text{ and } i \geq n \\ \mathrm{X'}^i & \text{otherwise.} \end{cases} \\
(\mathrm{m.X'}) \uparrow^n_{\mathrm{X}} & \stackrel{\text{def}}{=} & (\mathrm{m} \uparrow^n_{\mathrm{X}}).\mathrm{X'} \\
& \vdots & \\
(\mathbf{functor}(\mathrm{X'} : \mathrm{S})\mathrm{m}) \uparrow^n_{\mathrm{X}} & \stackrel{\text{def}}{=} & \begin{cases} \mathbf{functor}(\mathrm{X'} : (\mathrm{S} \uparrow^n_{\mathrm{X}}))(\mathrm{m} \uparrow^n_{\mathrm{X}}) & \text{if } \mathrm{X} \neq \mathrm{X'} \\ \mathbf{functor}(\mathrm{X'} : (\mathrm{S} \uparrow^n_{\mathrm{X}}))(\mathrm{m} \uparrow^{(n+1)}_{\mathrm{X}}) & \text{if } \mathrm{X} = \mathrm{X'} \end{cases} \\
& \vdots & \\
(\mathbf{type}\ \mathrm{t} = \mathrm{d}; \mathrm{b}) \uparrow^n_{\mathrm{X}} & \stackrel{\text{def}}{=} & \mathbf{type}\ \mathrm{t} = (\mathrm{d} \uparrow^n_{\mathrm{X}}); (\mathrm{b} \uparrow^n_{\mathrm{X}}) \\
& \vdots &
\end{array}$$

Lifting of type identifiers is analogous:

$$\begin{array}{rcl}
\mathrm{t'}^i \uparrow^n_{\mathrm{t}} & \stackrel{\text{def}}{=} & \begin{cases} \mathrm{t}^{i+1} & \text{if } \mathrm{t} = \mathrm{t'} \text{ and } i \geq n \\ \mathrm{t'}^i & \text{otherwise.} \end{cases} \\
(\mathrm{m.t'}) \uparrow^n_{\mathrm{t}} & \stackrel{\text{def}}{=} & (\mathrm{m} \uparrow^n_{\mathrm{t}}).\mathrm{t'} \\
& \vdots & \\
(\mathbf{functor}(\mathrm{X} : \mathrm{S})\mathrm{m}) \uparrow^n_{\mathrm{t}} & \stackrel{\text{def}}{=} & \mathbf{functor}(\mathrm{X} : (\mathrm{S} \uparrow^n_{\mathrm{t}}))(\mathrm{m} \uparrow^n_{\mathrm{t}}) \\
& \vdots & \\
(\mathbf{type}\ \mathrm{t'} = \mathrm{d}; \mathrm{b}) \uparrow^n_{\mathrm{t}} & \stackrel{\text{def}}{=} & \begin{cases} \mathbf{type}\ \mathrm{t'} = (\mathrm{d} \uparrow^n_{\mathrm{t}}); (\mathrm{b} \uparrow^n_{\mathrm{t}}) & \text{if } \mathrm{t} \neq \mathrm{t'} \\ \mathbf{type}\ \mathrm{t} = (\mathrm{d} \uparrow^n_{\mathrm{t}}); (\mathrm{b} \uparrow^{(n+1)}_{\mathrm{t}}) & \text{if } \mathrm{t} = \mathrm{t'} \end{cases} \\
& \vdots &
\end{array}$$

We can now define substitution. Our definition is derived from the standard one for pure de Bruijn terms. The operation $[\mathrm{m}/\mathrm{X}^n]\mathrm{p}$ substitutes the module expression m for any free occurrences of X^n within the phrase p, adjusting the indices on other free occurrences of X accordingly. Intuitively, when we encounter the phrase X^i we get one of the following cases: if $i = n$ then the substitution takes place; if $i > n$ then this is a free occurrence and must be adjusted to the free occurrence X^{i-1} (since the substitution eliminates one declaration of X); otherwise $i < n$ and this is a bound occurrence that remains unchanged. Note that if p is a declaration binding an identifier, then we need to prevent the capture of any free occurrences of that identifier in m. This is achieved by lifting references to that identifier in m, *before* substituting in the scope of the declaration. If the bound identifier happens to be X itself, then we also need to ensure that the substitution does not affect occurrences bound by *this* declaration of X. This is achieved by incrementing the index n of the substitution *before* descending into the scope of the declaration.

Definition 6.3 (Substitution). We will only show a few cases of the definition. The others are similar.

The substitution of a module expression for a module identifier is defined as follows:

$$
\begin{array}{rcl}
[\mathrm{m}/\mathrm{X}^n]\mathrm{X}'^i & \overset{\mathrm{def}}{=} & \begin{cases} \mathrm{m} & \text{if } \mathrm{X} = \mathrm{X}' \text{ and } i = n \\ \mathrm{X}^{i-1} & \text{if } \mathrm{X} = \mathrm{X}' \text{ and } i > n \\ \mathrm{X}'^i & \text{otherwise.} \end{cases} \\
[\mathrm{m}/\mathrm{X}^n](\mathrm{m}'.\mathrm{X}') & \overset{\mathrm{def}}{=} & ([\mathrm{m}/\mathrm{X}^n]\mathrm{m}').\mathrm{X}' \\
& \vdots & \\
[\mathrm{m}/\mathrm{X}^n](\mathbf{functor}(\mathrm{X}' : \mathrm{S})\mathrm{m}') & \overset{\mathrm{def}}{=} & \begin{cases} \begin{array}{l}\mathbf{functor}(\mathrm{X}' : ([\mathrm{m}/\mathrm{X}^n]\mathrm{S})) \\ \quad ([(\mathrm{m} \uparrow^0_{\mathrm{X}'})/\mathrm{X}^n]\mathrm{m}')\end{array} & \text{if } \mathrm{X} \neq \mathrm{X}' \\ \begin{array}{l}\mathbf{functor}(\mathrm{X}' : ([\mathrm{m}/\mathrm{X}^n]\mathrm{S})) \\ \quad ([(\mathrm{m} \uparrow^0_{\mathrm{X}'})/\mathrm{X}^{(n+1)}]\mathrm{m}')\end{array} & \text{if } \mathrm{X} = \mathrm{X}' \end{cases} \\
& \vdots & \\
[\mathrm{m}/\mathrm{X}^n](\mathbf{type}\ \mathrm{t} = \mathrm{d}; \mathrm{b}) & \overset{\mathrm{def}}{=} & \mathbf{type}\ \mathrm{t} = ([\mathrm{m}/\mathrm{X}^n]\mathrm{d}); ([(\mathrm{m} \uparrow^0_{\mathrm{t}})/\mathrm{X}^n]\mathrm{b}) \\
& \vdots &
\end{array}
$$

The substitution of a definable type for a type identifier is analogous:

$$
\begin{array}{rcl}
[\mathrm{d}/\mathrm{t}^n]\mathrm{t}'^i & \stackrel{\mathrm{def}}{=} & \begin{cases} \mathrm{d} & \text{if } \mathrm{t} = \mathrm{t}' \text{ and } i = n \\ \mathrm{t}^{i-1} & \text{if } \mathrm{t} = \mathrm{t}' \text{ and } i > n \\ \mathrm{t}'^i & \text{otherwise.} \end{cases} \\
[\mathrm{d}/\mathrm{t}^n](\mathrm{m}.\mathrm{t}') & \stackrel{\mathrm{def}}{=} & ([\mathrm{d}/\mathrm{t}^n]\mathrm{m}).\mathrm{t}' \\
& \vdots & \\
[\mathrm{d}/\mathrm{t}^n](\mathbf{functor}(\mathrm{X} : \mathrm{S})\mathrm{m}) & \stackrel{\mathrm{def}}{=} & \mathbf{functor}(\mathrm{X} : ([\mathrm{d}/\mathrm{t}^n]\mathrm{S}))([(\mathrm{d} \uparrow^0_{\mathrm{X}})/\mathrm{t}^n]\mathrm{m}) \\
& \vdots & \\
[\mathrm{d}/\mathrm{t}^n](\mathbf{type}\ \mathrm{t}' = \mathrm{d}'; \mathrm{b}) & \stackrel{\mathrm{def}}{=} & \begin{cases} \mathbf{type}\ \mathrm{t}' = ([\mathrm{d}/\mathrm{t}^n]\mathrm{d}'); ([(\mathrm{d} \uparrow^0_{\mathrm{t}'})/\mathrm{t}^n]\mathrm{b}) \\ \qquad\qquad \text{if } \mathrm{t} \neq \mathrm{t}' \\ \\ \mathbf{type}\ \mathrm{t}' = ([\mathrm{d}/\mathrm{t}^n]\mathrm{d}'); ([(\mathrm{d} \uparrow^0_{\mathrm{t}'})/\mathrm{t}^{(n+1)}]\mathrm{b}) \\ \qquad\qquad \text{if } \mathrm{t} = \mathrm{t}' \end{cases} \\
& \vdots &
\end{array}
$$

To motivate the definitions that follow, it is convenient to introduce some additional terminology. Let's define a signature expression to be *semantically complete* if, and only if, it denotes a semantic signature with an empty set of parameters. Note that a representation must always be semantically complete. Similarly, let's define a signature to be *syntactically complete* if, and only if, either it is a structure signature with a body that contains no type specifications of the form **type** t and only contains module specifications with syntactically complete signatures; or, it is a functor signature with a syntactically complete range. It is easy to verify that a signature expression can only be semantically complete if it is syntactically complete.

To establish the representation property, we will need to construct a syntactically complete signature for each module expression in order to obtain a representation of its type.

If the module expression is a projection m.X that is well-typed, then the representation of m's type must be a syntactically complete structure signature **sig** B **end**. Our method is to obtain a representation for m.X's type from the corresponding subsignature of B. Intuitively, the *projection* operation $\mathrm{B} \downarrow^{\mathrm{m}}_{\mathrm{X}}$, defined below, constructs a representation for the type of m.X from both m and its representation's body B. Note that we cannot merely extract the subsignature of X since it may contain occurrences of identifiers specified previously in the body. To prevent the introduction of dangling references, we substitute any occurrence of a previously specified type identifier by its definition, and any occurrence of a previously specified module component by the corresponding component of m. This is why the projection operation needs to take a module expression as an argument.

Definition 6.4 (Projection). Projection is defined as the following partial operation on signature bodies:

$$
\begin{array}{rcl}
(\mathbf{type}\ \mathrm{t};\mathrm{B}) \downarrow^{\mathrm{m}}_{\mathrm{X}} & \overset{\mathrm{def}}{=} & \textit{undefined} \\
(\mathbf{type}\ \mathrm{t} = \mathrm{d};\mathrm{B}) \downarrow^{\mathrm{m}}_{\mathrm{X}} & \overset{\mathrm{def}}{=} & [\mathrm{d}/\mathrm{t}^0](\mathrm{B} \downarrow^{(\mathrm{m}\uparrow^0_{\mathrm{t}})}_{\mathrm{X}}) \\
(\mathbf{module}\ \mathrm{X}' : \mathrm{S};\mathrm{B}) \downarrow^{\mathrm{m}}_{\mathrm{X}} & \overset{\mathrm{def}}{=} & \begin{cases} \mathrm{S} & \text{if } \mathrm{X} = \mathrm{X}' \\ [(\mathrm{m}.\mathrm{X}')/\mathrm{X}'^0](\mathrm{B} \downarrow^{(\mathrm{m}\uparrow^0_{\mathrm{X}'})}_{\mathrm{X}}) & \\ & \text{if } \mathrm{X} \neq \mathrm{X}' \end{cases} \\
\epsilon_{\mathrm{B}} \downarrow^{\mathrm{m}}_{\mathrm{X}} & \overset{\mathrm{def}}{=} & \textit{undefined}
\end{array}
$$

Note that if the signature **sig** B **end** is syntactically complete and B contains a specification of X, then the result of B $\downarrow^{\mathrm{m}}_{\mathrm{X}}$ is well-defined.

The final cases we need to deal with are when the module expression is an occurrence, say X^0, of an identifier declared as a functor argument **functor**(X : S)... X^0 ... or as a constrained definition **module** X : S = m′;... X^0 ..., or the module expression is a curtailment m ⪰ S. In each case, we would like to use the signature S as a representation but cannot because it may be syntactically incomplete. Our method of obtaining a representation is to complete the signature. This is the purpose of the *strengthening* operation, defined below. Strengthening the signature S by a matching module m, written S\m, converts every type specification of S into a definitional type specification derived from m, yielding a syntactically complete signature. In this way, the strengthened signature (S $\uparrow^0_{\mathrm{X}}$)\X^0 may be used as a representation of X^0's type; similarly, the strengthened signature S\m may be used as a representation of m ⪰ S's type.

Definition 6.5 (Strengthening). The strengthening operation is defined as follows.

$$\begin{array}{rcl}
(\textbf{type}\ \mathrm{t} = \mathrm{d}; \mathrm{B})\backslash\mathrm{m} & \overset{\text{def}}{=} & \textbf{type}\ \mathrm{t} = \mathrm{d}; (\mathrm{B}\backslash(\mathrm{m} \upharpoonleft^{0}_{\mathrm{t}})) \\
(\textbf{type}\ \mathrm{t};\mathrm{B})\backslash\mathrm{m} & \overset{\text{def}}{=} & \textbf{type}\ \mathrm{t} = \mathrm{m.t}; (\mathrm{B}\backslash(\mathrm{m} \upharpoonleft^{0}_{\mathrm{t}})) \\
(\textbf{module}\ \mathrm{X} : \mathrm{S}; \mathrm{B})\backslash\mathrm{m} & \overset{\text{def}}{=} & \textbf{module}\ \mathrm{X} : (\mathrm{S}\backslash\mathrm{m.X}); (\mathrm{B}\backslash(\mathrm{m} \upharpoonleft^{0}_{\mathrm{X}})) \\
\epsilon_{\mathrm{B}}\backslash\mathrm{m} & \overset{\text{def}}{=} & \epsilon_{\mathrm{B}} \\
 & & \\
(\textbf{sig}\ \mathrm{B}\ \textbf{end})\backslash\mathrm{m} & \overset{\text{def}}{=} & \textbf{sig}\ (\mathrm{B}\backslash\mathrm{m})\ \textbf{end} \\
(\textbf{funsig}(\mathrm{X}{:}\mathrm{S})\mathrm{S}')\backslash\mathrm{m} & \overset{\text{def}}{=} & \textbf{funsig}(\mathrm{X}{:}\mathrm{S})(\mathrm{S}'\backslash((\mathrm{m} \upharpoonleft^{0}_{\mathrm{X}})\ \mathrm{X}^{0}))
\end{array}$$

Remark 6.5.1. The definitions of projection and strengthening are generalisations of similar operations originally proposed by Leroy [Ler95, Ler94]. In Leroy's formulation, the operations' module argument is a *path*. Courant [Cou97b] also exploits generalised operations.

6.5.3 Properties

The proof of the representation property relies on the following lemmas, which we state without further proof.

The denotation of a signature is closed under realisation:

Lemma 6.6 (Closure under Realisation).
If $\mathcal{C} \vdash \mathrm{B} \triangleright \mathcal{L}$*, then* $\varphi(\mathcal{C}) \vdash \mathrm{B} \triangleright \varphi(\mathcal{L})$.
If $\mathcal{C} \vdash \mathrm{S} \triangleright \mathcal{L}$*, then* $\varphi(\mathcal{C}) \vdash \mathrm{S} \triangleright \varphi(\mathcal{L})$.

The denotation of a signature is preserved by enrichment of the context:

Lemma 6.7 (Enrichment).
If $\mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash \mathrm{S} \triangleright \mathcal{L}$ *and* $\mathcal{M}' \succeq \mathcal{M}$ *then* $\mathcal{C}[\mathrm{X} : \mathcal{M}'] \vdash \mathrm{S} \triangleright \mathcal{L}$.

The denotation of a signature can be preserved by lifting the signature past an additional declaration in the context:

Lemma 6.8 (Weakening).
If $\mathcal{C} \vdash \mathrm{S} \triangleright \mathcal{L}$ *then* $\mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash \mathrm{S} \upharpoonleft^{0}_{\mathrm{X}} \triangleright \mathcal{L}$ *and* $\mathcal{C}[\mathrm{t} = \tau] \vdash \mathrm{S} \upharpoonleft^{0}_{\mathrm{t}} \triangleright \mathcal{L}$.

The denotation of a signature or signature body is preserved by discharging a declaration with a module expression of the appropriate type:

Lemma 6.9 (Substitution). *Provided* $\mathcal{C} \vdash \mathrm{m} : \mathcal{M}$*:*

- $\mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash \mathrm{S} \triangleright \mathcal{L}$ *implies* $\mathcal{C} \vdash [\mathrm{m}/\mathrm{X}^{0}]\mathrm{S} \triangleright \mathcal{L}$.

$$\boxed{\vdash\ \mathcal{C}\ \mathbf{Rep}}$$

$$\frac{}{\vdash\ \epsilon_{\mathcal{C}}\ \mathbf{Rep}} \tag{R-1}$$

$$\frac{\vdash\ \mathcal{C}\ \mathbf{Rep} \quad \mathcal{C} \vdash \mathrm{d} \triangleright d}{\vdash\ \mathcal{C}[\mathrm{t} = d]\ \mathbf{Rep}} \tag{R-2}$$

$$\frac{\vdash\ \mathcal{C}\ \mathbf{Rep} \quad \mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{M}}{\vdash\ \mathcal{C}[\mathrm{X} : \mathcal{M}]\ \mathbf{Rep}} \tag{R-3}$$

Figure 6.8: The definition of representable contexts.

- $\mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash \mathrm{B} \triangleright \mathcal{L}$ *implies* $\mathcal{C} \vdash [\mathrm{m}/\mathrm{X}^0]\mathrm{B} \triangleright \mathcal{L}$.

The denotation of a projected signature is the same as the corresponding component of the body's denotation, provided the module expression used in the projection has the type of the enclosing body:

Lemma 6.10 (Projection).
If $\mathcal{C} \vdash \mathrm{B} \triangleright \Lambda\emptyset.\mathcal{S}$, $\mathcal{C} \vdash \mathrm{m} : \mathcal{S}$ *and* $\mathrm{X} \in \mathrm{Dom}(\mathcal{S})$ *then the projection* $\mathrm{B} \downarrow^{\mathrm{m}}_{\mathrm{X}}$ *is well-defined and* $\mathcal{C} \vdash \mathrm{B} \downarrow^{\mathrm{m}}_{\mathrm{X}} \triangleright \Lambda\emptyset.\mathcal{S}(\mathrm{X})$.

Finally, strengthening a signature by a matching module yields the corresponding realisation of the signature's denotation:

Lemma 6.11 (Strengthening).
If $\mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{M}$, $\mathcal{C} \vdash \mathrm{m} : \mathcal{M}'$, $\mathcal{M}' \succeq \varphi(\mathcal{M})$, *and* $\mathrm{Dom}(\varphi) = P$ *then* $\mathcal{C} \vdash \mathrm{S}\backslash\mathrm{m} \triangleright \Lambda\emptyset.\varphi(\mathcal{M})$.

The proof of the representation property relies on maintaining the following predicate on contexts as an invariant:

Definition 6.12 (Representable Contexts). The predicate $\vdash\ \mathcal{C}\ \mathbf{Rep}$, read "$\mathcal{C}$ is representable", is defined as the least relation closed under the rules in Figure 6.8.

Intuitively, a context is representable if, and only if, every semantic object in its range arises as the denotation of *some* type phrase, where the denotation is derived from the preceding assumptions in the context. Note that the meta-variables d, S and P of Rules (R-2) and (R-3) are existentially quantified, since they occur in the premises, but not the conclusions, of the rules. If a context is representable, then every occurrence of a module identifier declared in that context can be given a syntactic representation of its type by taking the original signature expression, lifting it past the subsequent declarations in the context and then strengthening the resulting signature by the occurrence itself.

We can now state the main theorem of this Chapter:

Theorem 6.13 (Representation). *Provided* $\vdash\ \mathcal{C}$ **Rep**, *if* $\mathcal{C} \vdash \mathrm{m} : \mathcal{M}$ *then there is some signature expression* S *such that* $\mathcal{C} \vdash \mathrm{S} \triangleright \Lambda\emptyset.\mathcal{M}$*.*

Proof (Representation). *We use strong induction on the rules defining the classification judgements to prove the statements:*

$$\mathcal{C} \vdash \mathrm{b} : \mathcal{S} \supset\ \vdash\ \mathcal{C}\ \textbf{Rep} \supset \exists \bar{\mathrm{B}}.\mathcal{C} \vdash \bar{\mathrm{B}} \triangleright \Lambda\emptyset.\mathcal{S},$$
$$\mathcal{C} \vdash \mathrm{m} : \mathcal{M} \supset\ \vdash\ \mathcal{C}\ \textbf{Rep} \supset \exists \bar{\mathrm{S}}.\mathcal{C} \vdash \bar{\mathrm{S}} \triangleright \Lambda\emptyset.\mathcal{M}.$$

Here are the cases:

$\boxed{13}$ *By strong induction we can assume the original premises:*

$$\mathcal{C} \vdash \mathrm{d} \triangleright d, \tag{1}$$

$$\mathcal{C}[\mathrm{t} = d] \vdash \mathrm{b} : \mathcal{S}, \tag{2}$$

$$\mathrm{t} \notin \mathrm{Dom}(\mathcal{S}) \tag{3}$$

as well as the induction hypothesis:

$$\vdash\ \mathcal{C}[\mathrm{t} = d]\ \textbf{Rep} \supset \exists \bar{\mathrm{B}}.\mathcal{C}[\mathrm{t} = d] \vdash \bar{\mathrm{B}} \triangleright \Lambda\emptyset.\mathcal{S}. \tag{4}$$

We need to show:

$$\vdash\ \mathcal{C}\ \textbf{Rep} \supset \exists \bar{\mathrm{B}}.\mathcal{C} \vdash \bar{\mathrm{B}} \triangleright \Lambda\emptyset.\mathrm{t} = d, \mathcal{S}.$$

Assume:

$$\vdash\ \mathcal{C}\ \textbf{Rep}. \tag{5}$$

From (5) *and premise* (1) *if follows that:*

$$\vdash\ \mathcal{C}[\mathrm{t} = d]\ \textbf{Rep}. \tag{6}$$

By induction hypothesis (4) *on* (6)*, we obtain a signature body* B *such that:*

$$\mathcal{C}[\mathrm{t} = d] \vdash \mathrm{B} \triangleright \Lambda\emptyset.\mathcal{S}. \tag{7}$$

Applying Rule (7) *to* (1), (7) *and* (3) *we can derive:*

$$\mathcal{C} \vdash \textbf{type}\ \mathrm{t} = \mathrm{d}; \mathrm{B} \triangleright \Lambda\emptyset.\mathrm{t} = d, \mathcal{S}. \tag{8}$$

Choosing $\bar{\mathrm{B}} \equiv \textbf{type}\ \mathrm{t} = \mathrm{d}; \mathrm{B}$ *gives the desired result.*

$\boxed{\textbf{14}}$ *By strong induction we can assume the original premises:*

$$\mathcal{C} \vdash \mathrm{m} : \mathcal{M}, \tag{1}$$

$$\mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash \mathrm{b} : \mathcal{S}, \tag{2}$$

$$\mathrm{X} \notin \mathrm{Dom}(\mathcal{S}) \tag{3}$$

as well as the induction hypotheses:

$$\vdash\ \mathcal{C}\ \textbf{Rep} \supset \exists\bar{\mathrm{S}}.\mathcal{C} \vdash \bar{\mathrm{S}} \triangleright \Lambda\emptyset.\mathcal{M}, \tag{4}$$

$$\vdash\ \mathcal{C}[\mathrm{X} : \mathcal{M}]\ \textbf{Rep} \supset \exists\bar{\mathrm{B}}.\mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash \bar{\mathrm{B}} \triangleright \Lambda\emptyset.\mathcal{S}. \tag{5}$$

We need to show:

$$\vdash\ \mathcal{C}\ \textbf{Rep} \supset \exists\bar{\mathrm{B}}.\mathcal{C} \vdash \bar{\mathrm{B}} \triangleright \Lambda\emptyset.\mathrm{X} : \mathcal{M}, \mathcal{S}.$$

Assume:

$$\vdash\ \mathcal{C}\ \textbf{Rep}. \tag{6}$$

By induction hypothesis (4) *on* (6)*, we obtain a signature expression* S *such that:*

$$\mathcal{C} \vdash \mathrm{S} \triangleright \Lambda\emptyset.\mathcal{M}. \tag{7}$$

From (6) *and* (7) *if follows that:*

$$\vdash\ \mathcal{C}[\mathrm{X} : \mathcal{M}]\ \textbf{Rep}. \tag{8}$$

By induction hypothesis (5) *on* (8), *we obtain a signature body* B *such that:*

$$\mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash \mathrm{B} \triangleright \Lambda\emptyset.\mathcal{S}. \tag{9}$$

Applying Rule (9) *to* (7), (9) *and* (3) *we can derive:*

$$\mathcal{C} \vdash \textbf{module}\ \mathrm{X} : \mathrm{S}; \mathrm{B} \triangleright \Lambda\emptyset.\mathrm{X} : \mathcal{M}, \mathcal{S}. \tag{10}$$

Choosing $\bar{\mathrm{B}} \equiv \textbf{module}\ \mathrm{X} : \mathrm{S}; \mathrm{B}$ *gives the desired result.*

$\boxed{\textbf{15}}$ *By strong induction we can assume the original premises:*

$$\mathcal{C} \vdash \mathrm{m} : \mathcal{M}, \tag{1}$$

$$\mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{M}', \tag{2}$$

$$\mathcal{M} \succeq \varphi\left(\mathcal{M}'\right), \tag{3}$$

$$\mathrm{Dom}(\varphi) = P, \tag{4}$$

$$P \cap \mathrm{FV}(\mathcal{C}) = \emptyset, \tag{5}$$

$$\mathcal{C}[\mathrm{X} : \mathcal{M}'] \vdash \mathrm{b} : \mathcal{S}, \tag{6}$$

$$\mathrm{X} \notin \mathrm{Dom}(\mathcal{S}) \tag{7}$$

as well as the induction hypotheses:

$$\vdash\ \mathcal{C}\ \textbf{Rep} \supset \exists\bar{\mathrm{S}}.\mathcal{C} \vdash \bar{\mathrm{S}} \triangleright \Lambda\emptyset.\mathcal{M}, \tag{8}$$

$$\vdash\ \mathcal{C}[\mathrm{X} : \mathcal{M}']\ \textbf{Rep} \supset \exists\bar{\mathrm{B}}.\mathcal{C}[\mathrm{X} : \mathcal{M}'] \vdash \bar{\mathrm{B}} \triangleright \Lambda\emptyset.\mathcal{S}. \tag{9}$$

We need to show:

$$\vdash\ \mathcal{C}\ \textbf{Rep} \supset \exists\bar{\mathrm{B}}.\mathcal{C} \vdash \bar{\mathrm{B}} \triangleright \Lambda\emptyset.(\mathrm{X} : \varphi\left(\mathcal{M}'\right), \varphi\left(\mathcal{S}\right)).$$

Assume:

$$\vdash\ \mathcal{C}\ \mathbf{Rep}. \tag{10}$$

By Lemma 6.11 (Strengthening) on (2), (1), (3) and (4) we have:

$$\mathcal{C} \vdash \mathrm{S}\backslash\mathrm{m} \triangleright \Lambda\emptyset.\varphi\left(\mathcal{M}'\right). \tag{11}$$

From (10) and (2) it is easy to show that:

$$\vdash\ \mathcal{C}[\mathrm{X} : \mathcal{M}']\ \mathbf{Rep}. \tag{12}$$

By induction hypothesis (9) on (12), we obtain a signature body B *such that:*

$$\mathcal{C}[\mathrm{X} : \mathcal{M}'] \vdash \mathrm{B} \triangleright \Lambda\emptyset.\mathcal{S}. \tag{13}$$

By Lemma 6.6 (Closure under Realisation) on φ and (13) we obtain:

$$\varphi\left(\mathcal{C}[\mathrm{X} : \mathcal{M}']\right) \vdash \mathrm{B} \triangleright \varphi\left(\Lambda\emptyset.\mathcal{S}\right). \tag{14}$$

By (4) and (5), (14) may be expressed as:

$$\mathcal{C}[\mathrm{X} : \varphi\left(\mathcal{M}'\right)] \vdash \mathrm{B} \triangleright \Lambda\emptyset.\varphi\left(\mathcal{S}\right). \tag{15}$$

Applying Rule (9) to (11), (15) and (7) we can derive:

$$\mathcal{C} \vdash \mathbf{module}\ \mathrm{X} : \mathrm{S}\backslash\mathrm{m}; \mathrm{B} \triangleright \Lambda\emptyset.\mathrm{X} : \varphi\left(\mathcal{M}\right), \varphi\left(\mathcal{S}\right). \tag{16}$$

Choosing $\bar{\mathrm{B}} \equiv \mathbf{module}\ \mathrm{X} : \mathrm{S}\backslash\mathrm{m}; \mathrm{B}$ *gives the desired result.*

$\boxed{\mathbf{16}}$ *By strong induction we can assume the original premises:*

$$\mathcal{C} \vdash \mathrm{m} : \mathcal{M}, \tag{1}$$

$$\mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash \mathrm{b} : \mathcal{S}, \tag{2}$$

as well as the induction hypotheses:

$$\vdash\ \mathcal{C}\ \mathbf{Rep} \supset \exists\bar{\mathrm{S}}.\mathcal{C} \vdash \bar{\mathrm{S}} \triangleright \Lambda\emptyset.\mathcal{M}, \tag{3}$$

$$\vdash\ \mathcal{C}[\mathrm{X} : \mathcal{M}]\ \mathbf{Rep} \supset \exists\bar{\mathrm{B}}.\mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash \bar{\mathrm{B}} \triangleright \Lambda\emptyset.\mathcal{S}. \tag{4}$$

We need to show:

$$\vdash \ \mathcal{C}\ \mathbf{Rep} \supset \exists \bar{\mathrm{B}}.\mathcal{C} \vdash \bar{\mathrm{B}} \rhd \Lambda\emptyset.\mathcal{S}.$$

Assume:

$$\vdash \ \mathcal{C}\ \mathbf{Rep}. \tag{5}$$

By induction hypothesis (3) *on* (5)*, we obtain a signature expression* S *such that:*

$$\mathcal{C} \vdash \mathrm{S} \rhd \Lambda\emptyset.\mathcal{M}. \tag{6}$$

From (5) *and* (6) *if follows that:*

$$\vdash \ \mathcal{C}[\mathrm{X} : \mathcal{M}]\ \mathbf{Rep}. \tag{7}$$

By induction hypothesis (4) *on* (7)*, we obtain a signature body* B *such that:*

$$\mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash \mathrm{B} \rhd \Lambda\emptyset.\mathcal{S}. \tag{8}$$

Using Lemma 6.9 (Substitution) on (1) *and* (8)*, we can substitute* m' *for* X *to obtain:*

$$\mathcal{C} \vdash [\mathrm{m}'/\mathrm{X}^0]\mathrm{B} \rhd \Lambda\emptyset.\mathcal{S}. \tag{9}$$

Choosing $\bar{\mathrm{B}} \equiv [\mathrm{m}'/\mathrm{X}^0]\mathrm{B}$ *gives the desired result.*

$\boxed{\mathbf{17}}$ *Trivial.*

$\boxed{\mathbf{18}}$ *We need to show:*

$$\vdash \ \mathcal{C}[\mathrm{X} : \mathcal{M}]\ \mathbf{Rep} \supset \exists \bar{\mathrm{S}}.\mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash \bar{\mathrm{S}} \rhd \Lambda\emptyset.\mathcal{M}$$

Assume:

$$\vdash \ \mathcal{C}[\mathrm{X} : \mathcal{M}]\ \mathbf{Rep}. \tag{1}$$

Inverting (1) *there must be some signature* S *and set of variables* P *such that:*

$$\mathcal{C} \vdash \mathrm{S} \rhd \Lambda P.\mathcal{M}. \tag{2}$$

By Lemma 6.8 (Weakening) on (2) *extended with the declaration of* X *we have:*

$$\mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash \mathrm{S} \uparrow^{0}_{\mathrm{X}} \rhd \Lambda P.\mathcal{M}. \tag{3}$$

By Rule (18) on (3) we can derive:

$$\mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash \mathrm{X}^0 : \mathcal{M}. \tag{4}$$

Let φ be the identity realisation with:

$$\mathrm{Dom}(\varphi) = P. \tag{5}$$

From reflexivity of $\_ \succeq \_$ it follows that:

$$\mathcal{M} \succeq \varphi(\mathcal{M}). \tag{6}$$

By Lemma 6.11 (Strengthening) on (3), (4) and (6) and (5) we obtain:

$$\mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash (\mathrm{S} \uparrow^0_{\mathrm{X}})\backslash \mathrm{X}^0 \triangleright \Lambda\emptyset.\varphi(\mathcal{M}).$$

Since φ is the identity, this is equivalent to:

$$\mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash (\mathrm{S} \uparrow^0_{\mathrm{X}})\backslash \mathrm{X}^0 \triangleright \Lambda\emptyset.\mathcal{M}. \tag{7}$$

Choosing $\bar{\mathrm{S}} \equiv (\mathrm{S} \uparrow^0_{\mathrm{X}})\backslash \mathrm{X}^0$ gives the desired result.

$\boxed{\mathbf{19}}$ *By strong induction we can assume the original premise:*

$$\mathcal{C} \vdash \mathrm{X}^n : \mathcal{M}, \tag{1}$$

as well as the induction hypothesis:

$$\vdash\ \mathcal{C}\ \mathbf{Rep} \supset \exists\bar{\mathrm{S}}.\mathcal{C} \vdash \bar{\mathrm{S}} \triangleright \Lambda\emptyset.\mathcal{M} \tag{2}$$

We need to show:

$$\vdash\ \mathcal{C}[\mathrm{X} : \mathcal{M}']\ \mathbf{Rep} \supset \exists\bar{\mathrm{S}}.\mathcal{C}[\mathrm{X} : \mathcal{M}'] \vdash \bar{\mathrm{S}} \triangleright \Lambda\emptyset.\mathcal{M}.$$

Assume

$$\vdash\ \mathcal{C}[\mathrm{X} : \mathcal{M}']\ \mathbf{Rep}. \tag{3}$$

Inverting (3) we must have:

$$\vdash\ \mathcal{C}\ \mathbf{Rep}. \tag{4}$$

By induction hypothesis (2) *on* (4) *we obtain a signature* S *such that:*

$$\mathcal{C} \vdash \mathrm{S} \triangleright \Lambda\emptyset.\mathcal{M}. \tag{5}$$

By Lemma 6.8 (Weakening) on (5) *extended with the declaration of* X *we obtain:*

$$\mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash \mathrm{S} \uparrow^{0}_{\mathrm{X}} \triangleright \Lambda\emptyset.\mathcal{M}. \tag{6}$$

Choosing $\bar{\mathrm{S}} \equiv \mathrm{S} \uparrow^{0}_{\mathrm{X}}$ *gives the desired result.*

$\boxed{\mathbf{20}}$ *Similar to case* $\boxed{\mathbf{\mathit{19}}}$, *except that we weaken by the declaration of a distinct module identifier, lifting references to this module identifier.*

$\boxed{\mathbf{21}}$ *Similar to case* $\boxed{\mathbf{\mathit{19}}}$, *except that we weaken by the declaration of a type identifier, lifting references to this type identifier.*

$\boxed{\mathbf{22}}$ *By strong induction we can assume the original premises:*

$$\mathcal{C} \vdash \mathrm{m} : \mathcal{S}, \tag{1}$$

$$\mathrm{X} \in \mathrm{Dom}(\mathcal{S}), \tag{2}$$

$$\mathcal{S}(\mathrm{X}) = \mathcal{M} \tag{3}$$

as well as the induction hypothesis:

$$\vdash\ \mathcal{C}\ \mathbf{Rep} \supset \exists\bar{\mathrm{S}}.\mathcal{C} \vdash \bar{\mathrm{S}} \triangleright \Lambda\emptyset.\mathcal{S} \tag{4}$$

We need to show:

$$\vdash\ \mathcal{C}\ \mathbf{Rep} \supset \exists\bar{\mathrm{S}}.\mathcal{C} \vdash \bar{\mathrm{S}} \triangleright \Lambda\emptyset.\mathcal{M}.$$

Assume:

$$\vdash\ \mathcal{C}\ \mathbf{Rep}. \tag{5}$$

By induction hypothesis (4) *on* (5) *we obtain a signature expression* S *such that:*

$$\mathcal{C} \vdash \mathrm{S} \triangleright \Lambda\emptyset.\mathcal{S}. \tag{6}$$

Inverting (6), *which can only have been derived by Rule* (11), *we must have* $\mathrm{S} \equiv \mathbf{sig}\ \mathrm{B}\ \mathbf{end}$ *for some signature body* B *such that:*

$$\mathcal{C} \vdash \mathrm{B} \triangleright \Lambda\emptyset.\mathcal{S}. \tag{7}$$

By Lemma 6.10 (Projection) applied to (7), and premises (1) and (2) we know that the projection $\mathrm{B} \downarrow^{\mathrm{m}}_{\mathrm{X}}$ *is well-defined and that:*

$$\mathcal{C} \vdash \mathrm{B} \downarrow^{\mathrm{m}}_{\mathrm{X}} \triangleright \Lambda\emptyset.\mathcal{S}(\mathrm{X}). \tag{8}$$

By (3) this can be re-expressed as:

$$\mathcal{C} \vdash \mathrm{B} \downarrow^{\mathrm{m}}_{\mathrm{X}} \triangleright \Lambda\emptyset.\mathcal{M}. \tag{9}$$

Choosing $\bar{\mathrm{S}} \equiv \mathrm{B} \downarrow^{\mathrm{m}}_{\mathrm{X}}$ *yields the desired result.*

$\boxed{\textbf{23}}$ *The result follows easily by induction on the premise.*

$\boxed{\textbf{24}}$ *By strong induction we can assume the original premises:*

$$\mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{M}, \tag{1}$$

$$P \cap \mathrm{FV}(\mathcal{C}) = \emptyset, \tag{2}$$

$$\mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash \mathrm{m} : \mathcal{M}' \tag{3}$$

as well as the induction hypothesis:

$$\vdash\ \mathcal{C}[\mathrm{X} : \mathcal{M}]\ \mathbf{Rep} \supset \exists\bar{\mathrm{S}}.\mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash \bar{\mathrm{S}} \triangleright \Lambda\emptyset.\mathcal{M}'. \tag{4}$$

We need to show:

$$\vdash\ \mathcal{C}\ \mathbf{Rep} \supset \exists\bar{\mathrm{S}}.\mathcal{C} \vdash \bar{\mathrm{S}} \triangleright \Lambda\emptyset.\forall P.\mathcal{M} \to \mathcal{M}'.$$

Assume:

$$\vdash\ \mathcal{C}\ \mathbf{Rep}. \tag{5}$$

From (5) and (1) we can derive:

$$\vdash\ \mathcal{C}[\mathrm{X} : \mathcal{M}]\ \mathbf{Rep}. \tag{6}$$

By induction hypothesis (4) on (6), we obtain a signature S' *such that:*

$$\mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash \mathrm{S}' \triangleright \Lambda\emptyset.\mathcal{M}'. \tag{7}$$

Applying Rule (12) to (1), (2) and (7) we can derive:

$$\mathcal{C} \vdash \mathbf{funsig}(\mathrm{X}{:}\mathrm{S})\mathrm{S}' \triangleright \Lambda\emptyset.\forall P.\mathcal{M} \to \mathcal{M}' \tag{8}$$

Choosing $\bar{\mathrm{S}} \equiv \mathbf{funsig}(\mathrm{X}{:}\mathrm{S})\mathrm{S}'$ *gives the desired result.*

$\boxed{\mathbf{25}}$ *By strong induction we can assume the original premises:*

$$\mathcal{C} \vdash \mathrm{m} : \forall Q.\mathcal{M}' \to \mathcal{M}, \tag{1}$$

$$\mathcal{C} \vdash \mathrm{m}' : \mathcal{M}'', \tag{2}$$

$$\mathcal{M}'' \succeq \varphi\left(\mathcal{M}'\right), \tag{3}$$

$$\mathrm{Dom}(\varphi) = Q \tag{4}$$

as well as the induction hypotheses:

$$\vdash\ \mathcal{C}\ \mathbf{Rep} \supset \exists \bar{\mathrm{S}}.\mathcal{C} \vdash \bar{\mathrm{S}} \triangleright \Lambda\emptyset.\forall Q.\mathcal{M}' \to \mathcal{M}, \tag{5}$$

$$\vdash\ \mathcal{C}\ \mathbf{Rep} \supset \exists \bar{\mathrm{S}}.\mathcal{C} \vdash \bar{\mathrm{S}} \triangleright \Lambda\emptyset.\mathcal{M}''. \tag{6}$$

We need to show:

$$\vdash\ \mathcal{C}\ \mathbf{Rep} \supset \exists \bar{\mathrm{S}}.\mathcal{C} \vdash \bar{\mathrm{S}} \triangleright \Lambda\emptyset.\varphi\left(\mathcal{M}\right).$$

Assume

$$\vdash\ \mathcal{C}\ \mathbf{Rep}. \tag{7}$$

By induction hypothesis (5) *on* (7)*, we obtain a signature* S *such that:*

$$\mathcal{C} \vdash \mathrm{S} \triangleright \Lambda\emptyset.\forall Q.\mathcal{M}' \to \mathcal{M}. \tag{8}$$

Inverting (8)*, which can only have been derived by Rule* (12)*, it follows that* $\mathrm{S} \equiv \mathbf{funsig}(\mathrm{X}{:}\mathrm{S}_1)\mathrm{S}_1$*, for some* S_1 *and* S_1*, such that:*

$$\mathcal{C} \vdash \mathrm{S}_1 \triangleright \Lambda Q.\mathcal{M}', \tag{9}$$

$$\mathcal{C}[\mathrm{X} : \mathcal{M}'] \vdash \mathrm{S}_2 \triangleright \Lambda\emptyset.\mathcal{M}, \tag{10}$$

where w.l.o.g. we can assume that:

$$Q \cap \mathrm{FV}(\mathcal{C}) = \emptyset. \tag{11}$$

By Lemma 6.6 (Closure under Realisation) applied to (10) and φ, we obtain:

$$\varphi\left(\mathcal{C}[\mathrm{X} : \mathcal{M}']\right) \vdash \mathrm{S}_2 \triangleright \varphi\left(\Lambda\emptyset.\mathcal{M}\right). \tag{12}$$

By (4) and (11) we can re-express (12) as:

$$\mathcal{C}[\mathrm{X} : \varphi\left(\mathcal{M}'\right)] \vdash \mathrm{S}_2 \triangleright \Lambda\emptyset.\varphi\left(\mathcal{M}\right). \tag{13}$$

By Lemma 6.7 (Enrichment) on (13) and (3) we obtain:

$$\mathcal{C}[\mathrm{X} : \mathcal{M}''] \vdash \mathrm{S}_2 \triangleright \Lambda\emptyset.\varphi\left(\mathcal{M}\right). \tag{14}$$

Using Lemma 6.9 (Substitution) on (2) and (14), we can substitute m' *for* X *to obtain:*

$$\mathcal{C} \vdash [\mathrm{m}'/\mathrm{X}^0]\mathrm{S}_2 \triangleright \Lambda\emptyset.\varphi\left(\mathcal{M}\right). \tag{15}$$

Choosing $\bar{\mathrm{S}} \equiv [\mathrm{m}'/\mathrm{X}^0]\mathrm{S}_2$ *gives the desired result.*

$\boxed{\mathbf{26}}$ *By strong induction we can assume the original premises:*

$$\mathcal{C} \vdash \mathrm{m} : \mathcal{M}', \tag{1}$$

$$\mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{M}, \tag{2}$$

$$\mathcal{M}' \succeq \varphi\left(\mathcal{M}\right), \tag{3}$$

$$\mathrm{Dom}(\varphi) = P \tag{4}$$

as well as the (redundant) induction hypothesis:

$$\vdash\ \mathcal{C}\ \mathbf{Rep} \supset \exists\bar{\mathrm{S}}.\mathcal{C} \vdash \bar{\mathrm{S}} \triangleright \Lambda\emptyset.\mathcal{M}' \tag{5}$$

We need to show:

$$\vdash\ \mathcal{C}\ \mathbf{Rep} \supset \exists\bar{\mathrm{S}}.\mathcal{C} \vdash \bar{\mathrm{S}} \triangleright \Lambda\emptyset.\varphi\left(\mathcal{M}\right).$$

Assume:

$$\vdash\ \mathcal{C}\ \mathbf{Rep}. \tag{6}$$

By Lemma 6.11 (Strengthening) applied to (2), (1), (3) and (4) we have:

$$\mathcal{C} \vdash \mathrm{S}\backslash\mathrm{m} \triangleright \Lambda\emptyset.\varphi\left(\mathcal{M}\right). \tag{7}$$

Choosing $\bar{\mathrm{S}} \equiv \mathrm{S}\backslash\mathrm{m}$ *yields the desired result.*

6.6 Conclusion

In this chapter, we have shown that Modules can support separate compilation. We analysed why the naive approach to separate compilation fails in Standard ML, placing the blame on an inappropriate choice of compilation unit. We suggested a better notion of compilation unit, based on abstractions. From a practical perspective, this approach seems perfectly acceptable.

However, from a theoretical perspective, the problems posed by eclipsed identifiers and anonymous abstract types mean that some module expressions may fail to possess syntactic representations of their types. We sketched a simplified (higher-order) Modules language that, by introducing indexed identifiers and removing the abstraction phrase, always admits syntactic representations. This property is captured by Theorem 6.13.

The language we proposed sports a weak form of abstraction (the constrained module definition) that supports separate compilation. Unfortunately, the loss of general abstractions means that it is impossible to isolate a module expression from its program context if that expression occurs, not at the top-level, but deeper within the program.

We should point out that the proof of Theorem 6.13 is *constructive*. In principle, the proof can be implemented in a compiler to report module types to the user as syntactic signatures, instead of semantic objects. Such an implementation can relieve the programmer of the burden of understanding semantic objects.

Chapter 7

First-Class Modules

In Chapter 5 we promoted the status of functors by making them first-class citizens of the Modules language. Although much more expressive than first-order Modules, Higher-Order Modules still maintains a rigid stratification between Modules and the Core. The notion of computation at the level of Modules is very weak, consisting solely of functor application, to model the linking of modules, and projection, to provide access to the components of structures. This weakness reflects the historical intention that Modules should merely be used to express the architecture of Core programs: actual algorithms and data structures are expressed by Core values and types.

To support general-purpose programming, the Core must provide more powerful notions of computation than Modules. For instance, Standard ML's Core supports recursive types and functions, control constructs, exceptions and references. Unfortunately, the stratification between Core and Modules means that the computational mechanisms of the Core cannot be exploited in the construction of modules. In this chapter, we relax this restriction by making modules *first-class* citizens of a particular Core language, Core-ML. In this extension, modules may be passed as arguments to Core-ML functions, returned as results of Core-ML computations, stored in data structures and so on.

In Section 7.1 we extend the grammar, semantic objects and static semantics of Core-ML to support first-class higher-order modules. In Section 7.2 we present an example illustrating the elegance of first-class modules. In Section 7.3 we give another example illustrating the additional expressive power of first-class modules. In Section 7.4 we propose an alternative, intuitively more natural elimination phrase for first-class modules but show that it is unsound. This violation of soundness highlights an important distinction between type abstraction at the level of Modules and its counterpart in the extended Core. In Section 7.4.1 we sketch a dynamic semantics for first-class modules and give a sketched proof that our static semantics is sound for this dynamic semantics. Section 7.5 closes with a brief assessment.

7.1 Core-ML with First-Class Modules

The motivation for introducing first-class modules is to extend the repertoire of computations producing module results. One way of achieving this end is to extend the class of module expressions and types directly with constructs usually associated with the Core. Taken to the extreme, this approach relaxes the stratification between Modules and the Core by removing it altogether: Modules and the Core are amalgamated in a single language. This is the route taken by Harper and Lillibridge [HL94], and explored further in the subsequent work by Lillibridge [Lil97], and Harper and Stone [SH96, HS97].

We adopt a different approach. We maintain the distinction between Core and Modules, but relax the stratification by enriching the Core language with a family of Core types, called *package types*, corresponding to first-class modules. A package type is introduced by encapsulating, or *packing*, a module as a Core expression. A package type is eliminated by breaking an encapsulation, *unpacking* an expression as a module in the scope of another expression. Because package types are ordinary Core types, packages are first-class citizens of the Core. The introduction and elimination phrases allow computation to alternate between computation at the level of Modules and computation at the level of the Core, without having to identify the notions of computation.

The advantage of preserving the distinction between Modules and the Core language is that we do not have to make an *a priori* commitment to a particular Core language in order to give the definition of Modules; the approach of amalgamating the Core and Modules into a single language, on the other hand, forces such a commitment from the outset. The advantage of distinguishing between Modules computation and Core computation is that each form of computation can be designed to satisfy different invariants. For instance, the invariant needed to support applicative functors, namely that the abstract types returned by a functor depend only on its type arguments and not the value of its term argument, is violated if we extend Modules computation directly with computational mechanisms such as conditional computation and recursion. On the other hand, these are precisely the sort of mechanisms that Core computation should provide. Applicative functors provide good support for programming with Higher-Order Modules; recursion and conditional computation are necessary in order to support realistic Core programs. By keeping Modules computation and Core computation separate, we can accommodate both. By contrast, although the amalgamated languages proposed by Harper and Lillibridge [HL94], Lillibridge [Lil97], and Harper and Stone [SH96, HS97] support higher-order functors, because there is only a single notion of computation, there is a trade-off between supporting either applicative functors or recursion and conditional computation. Since ruling out the latter is too severe a restriction, functors are not applicative.

From now on we will refer to a first-class module as a package, and its type as a package type. For concreteness, we will describe our extension of the Core with package types as an extension of a particular Core language, Core-ML. The technique should apply to other Core languages as well.

In Chapter 3, we were able give a presentation of Core-ML parameterised by an arbitrary Modules language, exploiting the fact that the points of contact between Core and Modules are few. This level of abstraction enabled us to generalise first-order Modules to Higher-Order Modules while keeping the definition of Core ML essentially fixed. Since the aim of this chapter is to extend Core-ML with first-class modules, we will need to make a stronger commitment to a particular modules language. For maximum generality, we will fix the modules language to be Higher-Order Modules.

We shall call the language resulting from the combination of Higher-Order Modules, Core-ML and package types *First-Class Modules*.

7.1.1 Phrase Classes

The grammar of Core-ML must be extended to support package types.

The grammar of Core-ML type phrases is modified by extending the grammar of simple types with the phrase $\langle \mathrm{S} \rangle \in \mathrm{SimTyp}$, specifying a package type. For instance, if we were to adopt a call-by-value dynamic semantics, then $\langle \mathrm{S} \rangle$ would specify the simple type of a Core value that encapsulates a module value, where the type of the module value must match the signature S. The denotation rule for package types will ensure that the denotation of the simple type $\langle \mathrm{S} \rangle$ is derived from the denotation of the encapsulated signature expression S.

The grammar of Core-ML value expressions is extended with phrases introducing and eliminating package types.

The value expression **pack** m **as** $\mathrm{S} \in \mathrm{ValExp}$ introduces a value of pack-

CoreId	$\overset{\text{def}}{=}$	$\{\mathbf{i}, \mathbf{j}, \ldots\}$	λ-bound identifiers
SimTypVar	$\overset{\text{def}}{=}$	$\{'a, 'b, \ldots\}$	simple type variables
DefKind	$\overset{\text{def}}{=}$	$\{0, 1, 2, 3, \ldots\}$	kinds (arities)
u	::=	$'a$	
	|	$u \to u'$	function type
	|	$do(u_0, \ldots, u_{k-1})$	type occurrence
	|	<S>	*package type*
d	::=	$\Lambda('a_0, \ldots, 'a_{k-1}).u$	parameterised simple type
v	::=	$\forall 'a_0, \ldots, 'a_{n-1}.u$	polymorphic simple type
e	::=	i	identifier
	|	λi.e	λ-abstraction
	|	e e′	application
	|	vo	value occurrence
	|	**pack** m **as** S	*package introduction*
	|	**open** e **as** X : S **in** e′	*package elimination*

Figure 7.1: Grammar of Core-ML Extended with Package Types

age type <S>. For instance, in a call-by-value dynamic semantics, the phrase is evaluated by evaluating the module expression m and encapsulating the resulting module value as a Core-ML value. The classification rule will ensure that the module expression matches the signature S, via some realisation. The signature determines the package type of the expression, and is also used to make the actual realisation of types in m abstract.

The phrase **open** e **as** X : S **in** e′ ∈ ValExp eliminates a value of package type <S>. For instance, in a call-by-value dynamic semantics, the expression e is evaluated to an encapsulated module value, the module value is bound to the module identifier X, and the value of the entire phrase is obtained by evaluating the client expression e′ in the current environment extended with the value of X. The classification rule will ensure that e has package type <S>, that X is assumed to have the type of an arbitrary realisation of S, and that the type of e′ does not depend on the realisation. Note that the explicit signature determines the package type of e.

Figure 7.1 summarises the grammar of Core-ML extended with package types (cf. Figure 3.11).

7.1.2 Semantic Objects

The semantic objects of Core-ML must be extended to include the semantic counterpart of package type phrases.

The denotation of a package type phrase is a semantic package type $<\exists P.\mathcal{M}> \in \mathit{SimTyp}$. Variables in P are bound in $\mathcal{M}$. By definition, we admit a package type $<\exists P.\mathcal{M}>$ only if it satisfies the following well-formedness condition: the corresponding signature $\Lambda P.\mathcal{M}$ must be solvable ($\vdash \Lambda P.\mathcal{M}$ **Slv**) in the sense of Definition 5.37. The motivation for this proviso will be explained in a moment.

As usual, we identify package types that are equivalent up to renamings of bound variables. Moreover, since we do not want to distinguish between package types that differ merely in a reordering of components, we from now on identify all package types that are equivalent according to the following definition:

Definition 7.1 (Equivalence of Package Types). Two package types $<\exists P.\mathcal{M}>$ and $<\exists P'.\mathcal{M}'>$ are *equivalent*, written $<\exists P.\mathcal{M}> = <\exists P'.\mathcal{M}'>$ if, and only if:

- $P' \cap \mathrm{FV}(\exists P.\mathcal{M}) = \emptyset$ and $\mathcal{M}' \succeq \varphi(\mathcal{M})$ for some φ with $\mathrm{Dom}(\varphi) = P$; and, symmetrically,
- $P \cap \mathrm{FV}(\exists P'.\mathcal{M}') = \emptyset$ and $\mathcal{M} \succeq \varphi'(\mathcal{M}')$ for some φ' with $\mathrm{Dom}(\varphi') = P'$.

The well-formedness condition on package types is intended to ensure that the equivalence on simple types is decidable. Intuitively, the equivalence of two well-formed package types can be decided by two invocations of the signature matching algorithm.

Conjecture 7.2 (Decidability). *The equivalence on package types of Definition 7.1 is decidable.*

Proof (Sketch). *Consider the two package types $<\exists P.\mathcal{M}>$ and $<\exists P'.\mathcal{M}'>$. Suppose we want to decide whether they are equivalent by verifying the conditions of Definition 7.1. Since P and P' are bound we can w.l.o.g. assume that $P' \cap \mathrm{FV}(\exists P.\mathcal{M}) = \emptyset$ and $P \cap \mathrm{FV}(\mathcal{M}') = \emptyset$. By the definition of well-formed package types we have $\vdash \Lambda P'.\mathcal{M}'$* **Slv**. *Two straightforward applications of Lemma 5.31 (Grounding) and Lemma 5.38 (Elimination) establish* *that* $\forall\mathrm{FV}(\mathcal{M}') \vdash \mathcal{M}'$ **Gnd**. *By the definition of well-formed package types we also have* $\vdash \Lambda P.\mathcal{M}$ **Slv**. *Appealing to Lemma 5.40 (Invocation) it follows that there exists a realisation* φ *such that* $\mathcal{M}' \succeq \varphi(\mathcal{M})$ *with* $\mathrm{Dom}(\varphi) = P$ *if, and only if, the invocation of the algorithm* $\forall\mathrm{FV}(\mathcal{M}') \cup \mathrm{FV}(\Lambda P.\mathcal{M}).\forall\emptyset \vdash \mathcal{M}' \succeq \mathcal{M} \downarrow \_$. *succeeds. Thus the first half of Definition 7.1 can be decided by the matching algorithm. A symmetric argument applies for deciding the second half.*

Remark 7.1.1. Strictly speaking, the naive argument used in the proof of Conjecture 7.2 does not constitute a valid proof. This is because the definitions of simple type equivalence, realisation and enrichment are now intertwined, while our proofs of the lemmas to which we appeal assumed they were not. However, the purpose of this chapter is not to develop the full meta-theory of First-Class Modules but merely to sketch a plausible proposal. We conjecture that the supporting lemmas still hold, but with modified proofs.

Remark 7.1.2. The motivation for focusing on the decidability of package type equivalence may not be clear. The reason decidability is important is that the *equivalence* between simple types is fundamental to the static semantics of Core-ML. For instance, a Core-ML function application can be classified only if type of the argument is *equivalent* to the function's domain type. This should be contrasted with the static semantics of Modules for which the enrichment relation on module types is fundamental. For instance, a functor application can be classified only if the type of the argument *enriches* (a realisation) of the functor's domain. Enrichment is a pre-order, not an equivalence relation. The static semantics of Core-ML and Modules are predicated on different notions of type comparison, type equivalence on the one hand, and type ordering on the other. It is fortunate that we can use the ordering of Modules to define an appropriate equivalence on package types.

Figure 7.2 summarises the semantic objects of Core-ML extended with package types (cf. Figure 3.13). Notice the proviso that package types are well-formed.

7.1.3 Static Semantics

We extend the judgements defining the static semantics of Core-ML by adding the following three rules:

$d^{\mathrm{k}} \in \mathit{DefTyp}^{\mathrm{k}}$	$::=$	$\Lambda({}'a_0, \ldots, {}'a_{\mathrm{k}-1}).u$	parameterised simple type
		(provided ${}'a_0, \ldots, {}'a_{\mathrm{k}-1}$ distinct)	
$v \in \mathit{ValTyp}$	$::=$	$\forall {}'a_0, \ldots, {}'a_{n-1}.u$	polymorphic simple type
		(provided ${}'a_1, \ldots, {}'a_{n-1}$ distinct)	
$u \in \mathit{SimTyp}$	$::=$	${}'a$	simple type variable
	$\mid$	$u \to u'$	function space
	$\mid$	$\nu^{\mathrm{k}}(u_0, \ldots, u_{\mathrm{k}-1})$	type name occurrence
	$\mid$	$<\exists P.\mathcal{M}>$	*package type*
		(provided $\vdash \Lambda P.\mathcal{M}$ **Slv**)	

$$C \in \mathit{CoreContext} \stackrel{\text{def}}{=} \left\{ \mathcal{C}_{\mathrm{i}} \cup \mathcal{C}_{'\mathrm{a}} \;\middle|\; \begin{array}{l} \mathcal{C}_{\mathrm{i}} \in \text{CoreId} \xrightarrow{\text{fin}} \mathit{SimTyp}, \\ \mathcal{C}_{'\mathrm{a}} \in \text{SimTypVar} \xrightarrow{\text{fin}} \mathit{SimTypVar} \end{array} \right\}$$

Figure 7.2: Semantic Objects of Core-ML Extended with Package Types

Denotation Rules

Simple Types

$\boxed{\mathcal{C} \vdash \mathrm{u} \rhd u}$

$$\frac{\mathcal{C} \vdash \mathrm{S} \rhd \Lambda P.\mathcal{M}}{\mathcal{C} \vdash <\mathrm{S}> \rhd <\exists P.\mathcal{M}>} \tag{P-1}$$

(P-1) Since the denotation of the signature expression S must be solvable by Lemma 5.42 (Invariance), the resulting package type is well-formed.

Classification Rules

Monomorphic Values

$\boxed{\mathcal{C} \vdash \mathrm{e} : u}$

$$\frac{\begin{array}{l} \mathcal{C} \vdash \mathrm{m} : \exists P'.\mathcal{M}' \\ \mathcal{C} \vdash \mathrm{S} \rhd \Lambda P.\mathcal{M} \\ P' \cap \mathrm{FV}(\Lambda P.\mathcal{M}) = \emptyset \\ \mathcal{M}' \succeq \varphi(\mathcal{M}) \\ \mathrm{Dom}(\varphi) = P \end{array}}{\mathcal{C} \vdash \textbf{pack}\ \mathrm{m}\ \textbf{as}\ \mathrm{S} : <\exists P.\mathcal{M}>} \tag{P-2}$$

$$\frac{\begin{array}{l}\mathcal{C} \vdash \mathrm{e} : <\exists P.\mathcal{M}> \\ \mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{M} \\ P \cap \mathrm{FV}(\mathcal{C}) = \emptyset \\ \mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash \mathrm{e}' : u \\ P \cap \mathrm{FV}(u) = \emptyset\end{array}}{\mathcal{C} \vdash \textbf{open}\ \mathrm{e}\ \textbf{as}\ \mathrm{X} : \mathrm{S}\ \textbf{in}\ \mathrm{e}' : u} \qquad \text{(P-3)}$$

(P-2) The static semantics of **pack** m **as** S is almost identical to the semantics of the abstraction phrase m \ S (Rule (H-21)). It too introduces an existential quantifier. The only difference is that the type of the phrase is an encapsulated existentially quantified module type. Notice also that abstracting the type of m by an explicit signature ensures that the resulting package type is well-formed (by Lemma 5.42 (Invariance)). This would not be the case were we to use the classification of m directly to derive the package type "$<\exists P'.\mathcal{M}'>$", since there is no guarantee that the corresponding signature $\Lambda P'.\mathcal{M}'$ is solvable (for a counter-example, let m be the module expression in Example 6.4.3).

(P-3) The rule requires that e is a package of the specified package type <S>. By assuming that X has the type of an *arbitrary* realisation of S, the rule ensures that the type of e′ is parametric in the package's actual realisation. Moreover, the side condition $P \cap \mathrm{FV}(u) = \emptyset$ prevents the type of e′ from depending on this realisation. Observe that the explicit signature S uniquely determines the simple type of the expression e (up to the equivalence of package types), while guaranteeing that it is well-formed (by Lemma 5.42 (Invariance)). This makes the type inference problem for Core-ML tractable: intuitively, it means that a type inference algorithm never has to *guess* the type of an expression that is used as a package.

Remark 7.1.3. The syntax **pack** m **as** S and **open** e **as** X : S **in** e′ is deliberately designed to evoke the phrases **pack** τ e **as** $\exists\alpha{:}\kappa.\tau'$ and **open** e **as** $\alpha{:}\kappa, \mathrm{x}{:}\tau$ **in** e′ associated with the higher-order existential types of Type Theory (Section 2.2.2). Indeed, their classification rules are very similar.

Let's compare Rule (P-2) with the existential introduction rule of Figure 2.14:

$$\frac{\mathrm{C} \vdash \exists\alpha{:}\kappa.\tau' : \star \quad \mathrm{C} \vdash \tau : \kappa \quad \mathrm{C} \vdash \mathrm{e} : [\tau/\alpha]\,(\tau')}{\mathrm{C} \vdash \textbf{pack}\ \tau\ \mathrm{e}\ \textbf{as}\ \exists\alpha{:}\kappa.\tau' : \exists\alpha{:}\kappa.\tau'}$$

Both rules introduce an existential quantifier. In Rule (P-2), the witness to this quantifier is the implicit realisation φ, while in this rule, it is the explicitly specified type τ. The signature S in the phrase **pack** m **as** S serves the same role as the template $\exists\alpha{:}\kappa.\tau'$ in the phrase **pack** τ e **as** $\exists\alpha{:}\kappa.\tau'$: each indicates the type occurrences that are to be made abstract in the actual types of m and e, respectively. The premise $\mathcal{C} \vdash \mathrm{S} \rhd \Lambda P.\mathcal{M}$, establishing the signature's denotation, corresponds to the well-formedness premise $\mathrm{C} \vdash \exists\alpha{:}\kappa.\tau' : \star$. Finally, the premises $\mathcal{C} \vdash \mathrm{m} : \exists P'.\mathcal{M}'$ and $\mathcal{M}' \succeq \varphi\,(\mathcal{M})$ ensure that the actual type of m is a realisation of the existential module type, much as the premise $\mathrm{C} \vdash \mathrm{e} : [\tau/\alpha]\,(\tau')$ ensures that the type of e is a substitution instance of the existential type. The only substantive differences in these rules is that Rule (P-2) can introduce an n-ary, not just a unary quantifier; moreover, Rule (P-2) incorporates an appeal to enrichment that corresponds to permitting e's actual type to be a subtype of $[\tau/\alpha]\,(\tau')$.

Now let us compare Rule (P-3) with the existential elimination rule of Figure 2.14:

$$\frac{\begin{array}{l}\mathrm{C} \vdash \exists\alpha{:}\kappa.\tau : \star \\ \mathrm{C} \vdash \mathrm{e} : \exists\alpha{:}\kappa.\tau \\ \mathrm{C}, \alpha : \kappa, \mathrm{x} : \tau \vdash \mathrm{e}' : \tau' \\ \mathrm{C} \vdash \tau' : \star\end{array}}{\mathrm{C} \vdash \textbf{open}\ \mathrm{e}\ \textbf{as}\ \alpha{:}\kappa, \mathrm{x}{:}\tau\ \textbf{in}\ \mathrm{e}' : \tau'}$$

Both rules eliminate an existential quantifier. The declaration X:S in the phrase **open** e **as** X : S **in** e′ serves the same role as the declarations $\alpha{:}\kappa, \mathrm{x}{:}\tau$ in the phrase **open** e **as** $\alpha{:}\kappa, \mathrm{x}{:}\tau$ **in** e′: each specifies the existential type of the opened term. In Rule (P-3), the hypothetical witness to the quantifier is the implicit set of type variables P, while in this rule, it is the explicitly specified type variable α. Both P and α must be fresh with respect to the type variables in the context, ensuring that the body of each phrase is parametric in the actual witness of the quantifier (to see that α must be fresh, it suffices to show that $\mathrm{C}, \alpha : \kappa, \mathrm{x} : \tau \vdash \mathrm{e}' : \tau'$ only if $\vdash \mathrm{C}, \alpha : \kappa, \mathrm{x} : \tau$ **valid**, from which it then follows that $\alpha \notin \mathrm{FV}(\mathrm{C})$). Finally, the side-condition $P \cap \mathrm{FV}(u) = \emptyset$ plays the same role as the premise $\mathrm{C} \vdash \tau' : \star$: each verifies that the type of the body is independent of the actual witness to the quantifier (the correspondence may be difficult to see at first, but $\alpha \notin \mathrm{FV}(\tau')$ is a simple consequence of applying a free variable lemma to the premise $\mathrm{C} \vdash \tau' : \star$, coupled with the previous observation that $\alpha \notin \mathrm{FV}(\mathrm{C})$). Of course, the rules differ in the sense that Rule (P-3) eliminates an n-ary, not a unary quantifier.

Remark 7.1.4. Even without the well-formedness condition on package types,

```
signature Stream =
  sig  type state: 0;
       val start: state;
       val next: state → state;
       val value: state → int
  end
```

Figure 7.3: The signature `Stream` of integer streams

as long as the current context $\mathcal{C}$ is ground, then, *provided the package types occurring in* $\mathcal{C}$ *are well-formed*, any additional package types arising from the phrases <S>, **pack** m **as** S and **open** e **as** X : S **in** e′ will be well-formed too. In each case, the semantic package type that is specified, introduced or eliminated is determined from the denotation of the explicit signature S. This denotation is guaranteed to be solvable by Lemma 5.42 (Invariance). However, it is preferable to impose the well-formedness condition on the *definition* of semantic package types, since this makes the above proviso on $\mathcal{C}$ redundant, while ensuring that any test of equivalence with a pre-existing package type in $\mathcal{C}$ will be decidable; this measure also prevents Rules (C-7) and (C-9) from silently introducing ill-formed package types.

7.2 An Example: The Sieve of Eratosthenes

We can illustrate the elegance of package types using a nice example adapted from Mitchell and Plotkin's exploration of existential types [MP88]. We take no credit for the example itself, nor for its explanation, which is paraphrased from [MP88].

The example is an implementation of the Sieve of Eratosthenes. The Sieve is an algorithm for enumerating prime numbers. Let *Sieve* be the enumeration $2, 3, 5, 7, 11 \ldots$ of primes.

We can think of an enumeration as a *stream*, or infinite list, of values. For this example, we will only need streams of integers.

In turn, we can represent a stream as a "process", defined by a set of internal *states*, a designated initial or *start* state, a transition function taking us from one state to the *next* state, and a specific *value* associated with each state. Reading the values off the process's sequence of states yields the stream.

Our implementation of *Sieve* uses packages to represent streams. The

```
val sift =
  λs.open s as S:Stream in
      pack
         struct
            val divisor = S.value S.start;
            val filter = fix λfilter.
                  λstate. if (divides divisor (S.value state))
                             then (filter (S.next state))
                             else state;
            type state = S.state;
            val start = filter S.start;
            val next = λstate.filter (S.next state);
            val value = S.value
         end
      as Stream;
```

Figure 7.4: The function `sift` implementing *sift*.

abbreviated signature `Stream`, defined in Figure 7.3, specifies the type of a structure implementing a stream[1]. The denotation of `Stream` is the following semantic signature:

$$\Lambda\alpha^0.(\mathbf{state} = \alpha, \mathbf{start} : \alpha, \mathbf{next} : \alpha \to \alpha, \mathbf{value} : \alpha \to \mathbf{int}).$$

Given a stream s, let $\mathit{sift}(s)$ be the substream of s consisting of those values not divisible by the initial value of s.

Figure 7.4 depicts an implementation `sift` of *sift* based on representing streams as packaged modules of signature `Stream`. The function `sift` takes a packaged stream, opens it, constructs the filtered stream and returns it as a package[2]. Our implementation of `sift` assumes the existence of a boolean test `divides` i j that returns true if, and only if, i is divisible by j with remainder zero. It is easy to see that `sift` has the type:

$$u \to u$$

where u is the package type:

$$u \equiv \,<\exists\alpha^0.(\mathbf{state} = \alpha, \mathbf{start} : \alpha, \mathbf{next} : \alpha \to \alpha, \mathbf{value} : \alpha \to \mathbf{int})>.$$

[1]We use a signature abbreviation for convenience only: every occurrence of the signature identifier `Stream` in the code that follows can be removed by in-lining its definition.

[2]The function *sift* can also be implemented as a functor.

```
module Sieve =
  struct
     type state = <Stream>;
     val start = pack
                    struct
                       type state = int;
                       val start = 2;
                       val next = succ;
                       val value = λstate.state
                    end
                 as Stream;
     val next = sift;
     val value = λstate.open state as S:Stream in
                            S.value S.start
  end \ Stream;
```

Figure 7.5: The implementation `Sieve` of *Sieve*.

If *start* is the stream $2, 3, 4, 5, 6, 7, 8, \ldots$, then the stream obtained by taking the initial value of each stream in the sequence of streams:

$$\begin{array}{lclllllllllll}
\mathit{start} & = & \mathbf{2}, & 3, & 4, & 5, & 6, & 7, & 8, & 9, & 10, & 11, & \ldots \\
\mathit{sift}(\mathit{start}) & = & & \mathbf{3}, & & 5, & & 7, & & 9, & & 11, & \ldots \\
\mathit{sift}(\mathit{sift}(\mathit{start})) & = & & & & \mathbf{5}, & & 7, & & & & 11, & \ldots \\
\mathit{sift}(\mathit{sift}(\mathit{sift}(\mathit{start}))) & = & & & & & & \mathbf{7}, & & & & 11, & \ldots \\
 & \vdots & & & & & & & & & & &
\end{array}$$

will itself yield the stream of primes $2, 3, 5, 7, \ldots$.

This is the intuition for constructing the Sieve of Eratosthenes: *Sieve* is implemented as a process representing the stream of primes. The states of *Sieve* are streams. The start state of *Sieve* is the stream *start* of all integers ≥ 2. The next state of *Sieve* is obtained by *sift*ing the current state. The value of each *Sieve* state is the first value of that state viewed as a stream.

Figure 7.5 shows an implementation of *Sieve* as the structure `Sieve`. The type `Sieve.state` is the type of packaged streams. The value `Sieve.start` is the packaged stream of all integers ≥ 2. The function `Sieve.next` `sift`'s a given state. The function `Sieve.value` returns the first value of a state (opened as a stream).

```
val nthstate = fix λnthstate.
                   λn.ifzero n
                        (Sieve.start)
                        (Sieve.next (nthstate (+ n (-1))));

val nthprime =  λn.Sieve.value (nthstate n);
```

Figure 7.6: The function `nthprime` implementing the mathematical function *nthprime*.

Note that `Sieve` also matches the signature `Stream`, reflecting our conceptualisation of *Sieve* as a stream constructed from streams. In particular, the type of the implementation of `Sieve`, before abstraction by `Stream` is:

$$\exists\emptyset.(\mathbf{state} = u, \mathbf{start} : u, \mathbf{next} : u \to u, \mathbf{value} : u \to \mathbf{int}),$$

where:

$$u \equiv <\exists\alpha^0.(\mathbf{state} = \alpha, \mathbf{start} : \alpha, \mathbf{next} : \alpha \to \alpha, \mathbf{value} : \alpha \to \mathbf{int})>.$$

Applying the abstraction yields:

$$\exists\alpha^0.(\mathbf{state} = \alpha, \mathbf{start} : \alpha, \mathbf{next} : \alpha \to \alpha, \mathbf{value} : \alpha \to \mathbf{int}).$$

(As an aside, notice that, because `Sieve` matches `Stream`, we can even apply the `sift` function to the package `pack Sieve as Stream`. Of course, in a call-by-value dynamic semantics, this application fails to terminate as the definition of the resulting start state diverges: since the values of `Sieve`'s states are distinct primes, none of the subsequent states of `Sieve` have a value that is divisible by the value of `Sieve`'s start state.)

In Figure 7.6, the value `nthprime` implements the mathematical function *nthprime*(n) that returns the n-th prime number, for $n \geq 0$. Notice how the function `nthstate` is used to construct the n-th state of the sieve, for an *arbitrary* n.

Besides conceptual elegance, what does the addition of package types really achieve in terms of computational power? Without package types, it is still possible to give a stratified implementation of *Sieve*, say as the structure `Sieve'` defined in Figure 7.7. `Sieve'` uses an ordinary structure `Start` for the initial state, a functor `Next` for computing state transitions, and another functor `Value` for extracting the value from a state.

Using `Sieve'`, we can still calculate the nth-prime with the expression:

```
module Sieve' =
  struct
     module Start = struct
                      type state = int;
                      val start = 2;
                      val next = succ;
                      val value = λstate.state
                   end;
     module Next = functor(S:Stream)
           struct
              type state  = S.state;
              val divisor = S.value S.start;
              val filter = fix λfilter.
                  λstate. if (divides divisor (S.value state))
                             then (filter (S.next state))
                             else state;
              val start = filter S.start;
              val next = λstate.filter (S.next state);
              val value = S.value
           end;
     module Value = functor(S:Stream)
                struct val value = S.value (S.start) end
  end
```

Figure 7.7: A stratified implementation `Sieve'` of *Sieve*.

```
signature Array =
 sig type array: 1;
     val init: ∀'a. 'a → (array 'a);
     val sub: ∀'a. (array 'a) → int → 'a;
     val update : ∀'a. (array 'a) → int → 'a → (array 'a)
 end;
```

Figure 7.8: The signature `Array` specifies a structure implementing fixed-size arrays.

```
(Sieve'.Value(Sieve'.Next(···Sieve'.Next(Sieve'.Start)···)))
.value
```

by chaining n applications of the functor `Sieve'.Next`. The problem is that we can only do this for a *fixed* n. This means that it is impossible to derive the function *nthprime* from `Sieve'` because there is no way to iterate the construction of `Sieve'`s intermediate states.

Remark 7.2.1. Of course, the Sieve of Eratosthenes can be implemented directly in Core-ML using other means. The point is that, without package types, we cannot use structures as a natural representation of streams.

Notice also that `Sieve'`, unlike `Sieve`, no longer matches the signature `Stream`, even though the states of `Sieve'` do. Thus the implementation `Sieve'` fails to capture the conceptualisation of *Sieve* as a stream constructed from streams. In the stratified language, it is impossible to capture this conceptualisation: if we use structures of signature *Stream* to represent the state's of the sieve, then the states of the structures may be represented using Core-ML values of a Core-ML type, but the state's of the sieve must be represented by Modules values of a Modules type.

7.3 Another Example: Dynamically-Sized Functional Arrays

With package types, it is perfectly possible to make the actual realisation of an abstract type depend on the result of some Core-ML computation. In this way, package types strictly extend the class of types that can be defined in Core-ML with Higher-Order Modules alone. Since this feature is not illustrated by our implementation of the Sieve of Eratosthenes, we will give a different example exploiting it here.

```
module ArrayZero=
  struct
     type array = Λ'a.'a;
     val init = λx.x;
     val sub = λa.λi.a;
     val update = λa.λi.λx.x
  end;
```

Figure 7.9: The structure `ArrayZero` implementing arrays of size 2^0.

```
module ArraySucc =
   functor(A:Array)
   struct
       type array = Λ'a.(A.array 'a) * (A.array 'a);
       val init = λx. pair (A.init x) (A.init x)
       val sub = λa.λi.
               ifzero (mod i 2)
                      (A.sub (fst a) (div i 2))
                      (A.sub (snd a) (div i 2));
       val update = λa.λi.λx.
              ifzero (mod i 2)
                 (pair (A.update (fst a) (div i 2) x) (snd a))
                 (pair (fst a) (A.update (snd a) (div i 2) x))
   end;
```

Figure 7.10: The functor `ArraySucc` mapping an implementation of arrays of size 2^n to an implementation of arrays of size 2^{n+1}.

```
val mkArray = fix (λmkArray.λn.
                  ifzero n
                  (pack ArrayZero as Array)
                  (open mkArray (+ n (-1)) as A:Array in
                     pack ArraySucc(A) as Array));
```

Figure 7.11: The function `mkArray`: applying `mkArray` to an integer $n \geq 0$ returns an abstract implementation of arrays of size 2^n.

A familiar example of a type whose representation depends on the result of some computation is the type of *dynamically* allocated arrays of size $n > 0$, where n is a value that is computed at run-time. To keep our example simple, we will implement *functional* arrays of size 2^n, for arbitrary $n \geq 0$.

Figure 7.8 defines the signature `Array` as a convenient abbreviation for the specification of a structure implementing polymorphic arrays. This signature should be interpreted as the following specification. For a fixed n, the type `array` u represents arrays containing 2^n entries of type u. The function `init` x creates an array that has its entries initialised to the value of x. The function `sub` a i returns the value of the (i mod 2^n)-th entry of the array. The function `update` a i x returns an array that is equivalent to the array a, except for the (i mod 2^n)-th entry that is updated with the value of x. Interpreting each index i modulo 2^n ensures that a client of an array never attempts to access or update a non-existent entry. We informally require that the functions `sub` and `update` have worst-case time-complexity $O(n)$; i.e. that their execution takes time logarithmic in the size of the array.

The structure `ArrayZero`, defined in Figure 7.9, is a trivial implementation of arrays of size $2^0 = 1$. An array is represented by the type of its sole entry. The function `init x` returns the initial value `x` of its entry, viewed as an array. Since (i mod 2^0) = 0, for any i, the function `sub` a i merely returns the value of the entire array a, viewed as an entry. Similarly, the function `sub` a i x updates the array a by simply returning the updated entry x, viewed as an array.

The functor `ArraySucc`, defined in Figure 7.10, maps a structure `A`, implementing arrays of size 2^n, to a structure implementing arrays of size 2^{n+1}. The definition of `ArraySucc` assumes that Core-ML has been extended with the cross product type, written u `*` u′, supporting pairing and projection:

```
pair : ∀'a 'b.  'a → 'b → ('a * 'b),
 fst : ∀'a 'b.  ('a * 'b) → 'a,
 snd : ∀'a 'b.  ('a * 'b) → 'b.
```

We additionally assume that the function `div` i j returns the largest integral divisor of i by j; and that the function `mod` i j returns the integral remainder of dividing i by j.

The functor `ArraySucc` represents an array of size 2^{n+1} as a pair of arrays of size 2^n. Entries with even indices are stored in the first component of the pair. Entries with odd indices are stored in the second component of the pair. The function `init` a returns a pair of arrays of size 2^n, initialised using the function `A.init` on arrays of size 2^n. The function `sub` a i uses the parity of i to determine which component of the array to inspect, returning its `(div i 2)`-th entry using the function `A.sub` on arrays of size 2^n. The function `update` a i x uses the parity of i to determine which component of the array to update, returning the pair of the unaltered component and the result of updating the other component using the function `A.update` on arrays of size 2^n. It is easy to see that `sub` and `update` are of time-complexity $O(n+1)$, provided `A.sub` and `A.update` are of complexity $O(n)$.

Figure 7.11 shows the definition of the Core-ML function `mkArray`. When applied to an integer n, it returns a package implementing arrays of size 2^n (provided $n \geq 0$). If $n = 0$, it simply returns the packaged structure `ArrayZero`. If $n \neq 0$, it first creates a package of arrays of size 2^{n-1} by recursion on $n-1$, and then uses this package to implement a package of arrays of size 2^n by a simple application of the functor `ArraySucc`. Notice that the actual realisation of the type of arrays returned by `mkArray` depends on the value of n. It is easy to reason that `mkArray` returns an implementation whose `sub` and `update` functions have complexity $0(n)$.

7.4 Soundness and Package Elimination

The phrase **pack** m **as** S turns a module expression into a value expression of the Core language. It is natural to expect this phrase to have a direct inverse that turns a value expression of package type back into a module expression. For instance, we might consider adding the module expression **unpack** e **as** S, with the classification rule:

$$\frac{\mathcal{C} \vdash \mathrm{e} : <\exists P.\mathcal{M}> \quad \mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{M}}{\mathcal{C} \vdash \textbf{unpack}\ \mathrm{e}\ \textbf{as}\ \mathrm{S} : \exists P.\mathcal{M}} \qquad (\star)$$

Unfortunately, combining this phrase with *applicative* functors is *unsound.*

Example 7.4.1. Consider the well-typed, but unsound counter-example in Figure 7.12. In the definition of the functor `F`, by exploiting the phrase **unpack** e **as** S we have made the implementation of the functor body conditional on the value of its argument's boolean component.

In particular, each branch of the conditional expression has package type:

$$<\exists \alpha^0.(\mathbf{t} = \alpha, \mathbf{x} : \alpha, \mathbf{y} : \alpha \to \alpha)>.$$

The standard classification rule for conditionals allows us to derive that the conditional expression also has this type. By unpacking this expression, we can give the functor body the existential module type:

$$\exists \alpha^0.(\mathbf{t} = \alpha, \mathbf{x} : \alpha, \mathbf{y} : \alpha \to \alpha).$$

```
module F =
  functor(X:sig val b:bool end)
      unpack if X.b
             then pack struct type t = int;
                              val x = 1;
                              val y = λx. -x
                        end
                  as sig type t:0;
                         val x:t;
                         val y:t → t
                      end
             else pack struct type t = bool;
                              val x = true;
                              val y = λx.if x then false
                                              else true
                        end
                  as sig type t:0;
                         val x:t;
                         val y:t → t
                      end
      as sig type t:0;
             val x:t;
             val y:t → t
         end;

module A = F (struct val b = true end);
module B = F (struct val b = false end);

val z = A.y B.x
```

Figure 7.12: Unpacking a Core value as a module is unsound.

Now, because functors are applicative, the type derived for the functor expression is:

$$\exists\alpha^0.\forall\emptyset.(\mathbf{b} : \mathbf{bool}) \to (\mathbf{t} = \alpha, \mathbf{x} : \alpha, \mathbf{y} : \alpha \to \alpha).$$

This type is *unsound*. Intuitively, it expresses that the abstract type returned by the functor is constant, i.e. that it returns the same type α irrespective of the value of its module argument. For this functor, this is patently false, since the realisation of α is conditional on the actual value of the functor argument.

To see how this can lead to a type violation, consider the remaining definitions in Figure 7.12. Because the definition of `F` eliminates the existential quantifier in the type of the functor expression, the abstract types `A.t` and `B.t` will be the same, causing the definition of `z` to type-check, even though it leads to the sad attempt of applying integer negation (`A.y`) to a boolean value (`B.x`).

In the functor introduction rule (Rule (H-18)), a functor is made applicative by raising the existential quantification in its body's type to a position quantifying the type of the functor itself. The rule ensures that each occurrence of an existentially quantified variable is parameterised by the type parameters of the functor. This skolemisation step is sound, provided the actual realisation of these variables has the property that it depends at most on the functor's type parameters, but not on the value of its actual argument. For Higher-Order Modules, it can be shown that this property holds as an invariant of the classification rules. Our example demonstrates how the addition of the phrase **unpack** e **as** S can violate the invariant.

Intuitively, the weaker elimination phrase **open** e **as** X : S **in** e′ remains sound because it manages to preserve the invariant. Since it only allows us to eliminate a package type in the scope of another value expression, its existentially quantified variables will never be skolemised by the functor introduction rule.

Another way to look at this is to consider the difference in the interpretation of the quantifiers in the judgements:

$$\mathcal{C} \vdash \mathrm{m} : \exists P.\mathcal{M},$$

$$\mathcal{C} \vdash \mathrm{e} : <\exists P.\mathcal{M}>.$$

In the first judgement, the module type $\exists P.\mathcal{M}$ hides a realisation of P that can depend at most on the *static* interpretation of the type variables occurring in $\mathcal{C}$. In the second judgement, the package type $<\exists P.\mathcal{M}>$ hides a realisation of P that can depend both on the static interpretation of the type variables in $\mathcal{C}$ and on the *dynamic* interpretation of the module and value identifiers in $\mathcal{C}$. It is sound to treat a purely static realisation as if it had a vacuous dynamic dependency. This forward direction justifies the soundness of the package introduction rule. It is not sound to treat a possibly dynamic realisation as if it were purely static. This explains why the stronger elimination rule for **unpack** e **as** S is unsound.

7.4.1 Towards a Proof of Type Soundness

To demonstrate that the proposal in this chapter is sound, we need to define a dynamic semantics for First-Class Modules and then prove that evaluating well-typed module and value expressions does not lead to type violations. Although a thorough treatment of the dynamic semantics takes us beyond the scope of this thesis, in this section, we will give a brief sketch of how this might be done. A fuller description may be found in Maharaj and Gunter's work on the definition of a dynamic semantics for higher-order Standard ML Modules [MG93].

Suppose we adopt a call-by-value semantics for First-Class Modules. One way to define such a semantics, akin to the formulation of the dynamic semantics of Standard ML, is to define a set of *core values* $\mathrm{v} \in \mathrm{CoreVal}$, that includes encapsulated module values $<\mathrm{V}>$ and function closures (whose form we shall leave unspecified); and a set of *module values* $\mathrm{V} \in \mathrm{ModVal}$ including functor closures $<\mathrm{X}, \mathcal{E}, \mathrm{m}>$ and structure values (whose form we shall also leave unspecified). The component $\mathcal{E}$ in a closure is a dynamic environment mapping module and value identifiers to values. We can then define an evaluation relation relating value expressions to their core values:

$$\boxed{\mathcal{E} \vdash \mathrm{e} \downarrow \mathrm{v}}$$

$$\vdots$$

$$\frac{\mathcal{E} \vdash \mathrm{m} \downarrow \mathrm{V}}{\mathcal{E} \vdash \textbf{pack}\ \mathrm{m}\ \textbf{as}\ \mathrm{S} \downarrow <\mathrm{V}>}$$

$$\frac{\mathcal{E} \vdash \mathrm{e} \downarrow <\mathrm{V}> \quad \mathcal{E}[\mathrm{X} = \mathrm{V}] \vdash \mathrm{e}' \downarrow \mathrm{v}}{\mathcal{E} \vdash \textbf{open}\ \mathrm{e}\ \textbf{as}\ \mathrm{X} : \mathrm{S}\ \textbf{in}\ \mathrm{e}' \downarrow \mathrm{v}}$$

$$\vdots$$

and Module expressions to their module values:

$$\boxed{\mathcal{E} \vdash \mathrm{m} \downarrow \mathrm{V}}$$

$$\vdots$$

$$\frac{}{\mathcal{E} \vdash \textbf{functor}(\mathrm{X} : \mathrm{S})\mathrm{m} \downarrow \texttt{<}\mathrm{X}, \mathcal{E}, \mathrm{m}\texttt{>}}$$

$$\frac{\mathcal{E} \vdash \mathrm{m} \downarrow \texttt{<}\mathrm{X}, \mathcal{E}', \mathrm{m}''\texttt{>} \quad \mathcal{E} \vdash \mathrm{m}' \downarrow \mathrm{V}' \quad \mathcal{E}'[\mathrm{X} = \mathrm{V}'] \vdash \mathrm{m}'' \downarrow \mathrm{V}''}{\mathcal{E} \vdash \mathrm{m}\ \mathrm{m}' \downarrow \mathrm{V}''}$$

$$\vdots$$

To prove that the static semantics is sound for the dynamic semantics, we can introduce a semantic classification judgement relating core values to their simple types:

$$\boxed{\vdash \mathrm{v} : u}$$

$$\vdots$$

$$\frac{\mathrm{Dom}(\varphi) = P \quad \vdash \mathrm{V} : \varphi\,(\mathcal{M})}{\vdash \texttt{<}\mathrm{V}\texttt{>} : \texttt{<}\exists P.\mathcal{M}\texttt{>}}$$

$$\vdots$$

and module values to their module types:

$$\boxed{\vdash \mathrm{V} : \mathcal{M}}$$

$$\vdots$$

$$\frac{\begin{array}{l} \forall\varphi.\mathrm{Dom}(\varphi) = P \supset \\ \quad \forall \mathrm{V}.\vdash \mathrm{V} : \varphi\,(\mathcal{M}) \supset \\ \quad\quad \forall \mathrm{V}'.\mathcal{E}[\mathrm{X} = \mathrm{V}] \vdash \mathrm{m} \downarrow \mathrm{V}' \supset \\ \quad\quad\quad \vdash \mathrm{V}' : \varphi\,(\mathcal{M}') \end{array}}{\vdash \texttt{<}\mathrm{X}, \mathcal{E}, \mathrm{m}\texttt{>} : \forall P.\mathcal{M} \to \mathcal{M}'}$$

$$\vdots$$

We say that an environment $\mathcal{E}$ has type $\mathcal{C}$, written $\vdash \mathcal{E} : \mathcal{C}$, if, and only if, every module and value identifier declared with a type in the context $\mathcal{C}$ is assigned a value in $\mathcal{E}$ that inhabits this type.

The type soundness property can then be stated as:

Property 7.3 (Type Soundness).

- $\mathcal{C} \vdash \mathrm{e} : u \supset$
  $\quad \vdash \mathcal{E} : \mathcal{C} \supset$
  $\quad\quad \mathcal{E} \vdash \mathrm{e} \downarrow \mathrm{v} \supset$
  $\quad\quad\quad \vdash \mathrm{v} : u.$

$$\vdots$$

- $\mathcal{C} \vdash \mathrm{m} : \exists P.\mathcal{M} \supset$
  $\quad \vdash \mathcal{E} : \mathcal{C} \supset$
  $\quad\quad \mathcal{E} \vdash \mathrm{m} \downarrow \mathrm{V} \supset$
  $\quad\quad\quad \exists \varphi.\ \mathrm{Dom}(\varphi) = P \wedge \vdash \mathrm{V} : \varphi(\mathcal{M}).$

Proof (Sketch). *Now to prove Property 7.3, we need to prove the stronger properties:*

- $\mathcal{C} \vdash \mathrm{e} : u \supset$
  $\quad \forall \psi, \sigma, \mathcal{E}, \mathrm{v}.$
  $\quad\quad \vdash \mathcal{E} : \psi(\sigma(\mathcal{C})) \supset$
  $\quad\quad\quad \mathcal{E} \vdash \mathrm{e} \downarrow \mathrm{v} \supset$
  $\quad\quad\quad\quad \vdash \mathrm{v} : \psi(\sigma(u)).$

$$\vdots$$

- $\mathcal{C} \vdash \mathrm{m} : \exists P.\mathcal{M} \supset$
  $\quad \exists \varphi.\mathrm{Dom}(\varphi) = P \wedge$
  $\quad\quad \forall \psi, \sigma, \mathcal{E}, \mathrm{V}.$
  $\quad\quad\quad \vdash \mathcal{E} : \psi(\sigma(\mathcal{C})) \supset$
  $\quad\quad\quad\quad \mathcal{E} \vdash \mathrm{m} \downarrow \mathrm{V} \supset$
  $\quad\quad\quad\quad\quad \vdash \mathrm{V} : \psi(\sigma(\varphi(\mathcal{M}))).$

by simultaneous induction on the classification rules.

In both properties, ψ is a realisation of type variables and σ is a substitution for simple type variables. Quantifying over all ψ and σ allows us to prove that the rules introducing type polymorphism and simple type polymorphism are sound. The second property tells us that the actual realisation φ of the module's abstract types P may vary uniformly with the static interpretations ψ and σ of type variable in the context $\mathcal{C}$, but not with the dynamic interpretation $\mathcal{E}$ of $\mathcal{C}$. Notice that this is strictly stronger than the requirement on φ in Property 7.3: in Property 7.3, φ is quantified within the scope of the dynamic environment $\mathcal{E}$, permitting it to vary with $\mathcal{E}$. The *stronger property of φ is needed in order to show that the skolemisation of existential types in the functor introduction rule (Rule (H-18)) is sound.*

To give an indication of how the proof proceeds, we will give the proof of type soundness for Rules (P-3) and (H-18). This proof remains a sketch because we have not verified the other cases, nor have we formalised the machinery necessary to do so.

$\boxed{\textbf{P-3}}$ *By induction we may assume:*

$$\begin{array}{l}\forall\psi,\sigma,\mathcal{E},\mathrm{v}.\\ \quad\vdash\mathcal{E}:\psi\,(\sigma(\mathcal{C}))\supset\\ \qquad\mathcal{E}\vdash\mathrm{e}\downarrow\mathrm{v}\supset\\ \qquad\quad\vdash\mathrm{v}:\psi\,(\sigma(<\exists P.\mathcal{M}>)),\end{array}\tag{1}$$

$$\mathcal{C}\vdash\mathrm{S}\rhd\Lambda P.\mathcal{M},\tag{2}$$

$$P\cap\mathrm{FV}(\mathcal{C})=\emptyset,\tag{3}$$

$$\begin{array}{l}\forall\psi,\sigma,\mathcal{E},\mathrm{v}.\\ \quad\vdash\mathcal{E}:\psi\,(\sigma(\mathcal{C}[\mathrm{X}:\mathcal{M}]))\supset\\ \qquad\mathcal{E}\vdash\mathrm{e}'\downarrow\mathrm{v}\supset\\ \qquad\quad\vdash\mathrm{v}:\psi\,(\sigma(u)),\end{array}\tag{4}$$

$$P\cap\mathrm{FV}(u)=\emptyset.\tag{5}$$

We need to show:

$$\begin{array}{l}\forall\psi,\sigma,\mathcal{E},\mathrm{v}.\\ \quad\vdash\mathcal{E}:\psi\,(\sigma(\mathcal{C}))\supset\\ \qquad\mathcal{E}\vdash\mathbf{open}\;\mathrm{e}\;\mathbf{as}\;\mathrm{X}:\mathrm{S}\;\mathbf{in}\;\mathrm{e}'\downarrow\mathrm{v}\supset\\ \qquad\quad\vdash\mathrm{v}:\psi\,(\sigma(u)).\end{array}$$

Consider arbitrary ψ, σ, $\mathcal{E}$ and v such that:

$$\vdash\mathcal{E}:\psi\,(\sigma(\mathcal{C})),\tag{6}$$

$$\mathcal{E}\vdash\mathbf{open}\;\mathrm{e}\;\mathbf{as}\;\mathrm{X}:\mathrm{S}\;\mathbf{in}\;\mathrm{e}'\downarrow\mathrm{v}.\tag{7}$$

We need to show:

$$\vdash\mathrm{v}:\psi\,(\sigma(u)).$$

W.l.o.g. we can assume:

$$P \cap (\mathrm{Inv}(\psi) \cup \mathrm{FV}(\sigma)) = \emptyset. \tag{8}$$

Inverting (7) *by the evaluation rule for* **open** e **as** X : S **in** e′ *we must have, for some module value* V*:*

$$\mathcal{E} \vdash \mathrm{e} \downarrow \texttt{<}\mathrm{V}\texttt{>}, \tag{9}$$

$$\mathcal{E}[\mathrm{X} = \mathrm{V}] \vdash \mathrm{e}' \downarrow \mathrm{v}. \tag{10}$$

By induction hypothesis (1) *applied to* ψ*,* σ*,* $\mathcal{E}$*,* <V>*,* (6) *and* (9) *we obtain:*

$$\vdash \texttt{<}\mathrm{V}\texttt{>} : \psi\,(\sigma(\texttt{<}\exists P.\mathcal{M}\texttt{>})), \tag{11}$$

which, by assumption(8)*, may be re-expressed as:*

$$\vdash \texttt{<}\mathrm{V}\texttt{>} : \texttt{<}\exists P.\psi\,(\sigma(\mathcal{M}))\texttt{>}. \tag{12}$$

Inverting (12) *by the classification rule for encapsulated module values, we must have some realisation* φ *such that:*

$$\mathrm{Dom}(\varphi) = P, \tag{13}$$

$$\vdash \mathrm{V} : \varphi\,(\psi\,(\sigma(\mathcal{M}))). \tag{14}$$

Let $\psi' = \psi \mid \varphi$*. Then, by* (13)*,* (3) *and* (8)*, we have:*

$$\psi'\,(\sigma(\mathcal{C})) = \psi\,(\sigma(\mathcal{C}))\,. \tag{15}$$

Moreover, by (13) *and* (8) *we have:*

$$\psi'\,(\sigma(\mathcal{M})) = \varphi\,(\psi\,(\sigma(\mathcal{M})))\,. \tag{16}$$

Combining (6) *and* (14) *we can show:*

$$\vdash \mathcal{E}[\mathrm{X} = \mathrm{V}] : (\psi\,(\sigma(\mathcal{C})))[\mathrm{X} : \varphi\,(\psi\,(\sigma(\mathcal{M})))],$$

which, by (15) *and* (16)*, may be expressed as:*

$$\vdash \mathcal{E}[\mathrm{X} = \mathrm{V}] : \psi'\,(\sigma(\mathcal{C}[\mathrm{X} : \mathcal{M}])). \tag{17}$$

By induction hypothesis (4) *on* ψ', σ, $\mathcal{E}[\mathrm{X} = \mathrm{V}]$, v, (17) *and* (10) *we obtain:*

$$\vdash \mathrm{v} : \psi'\,(\sigma(u)). \tag{18}$$

Now $\psi'\,(\sigma(u)) = \varphi\,(\psi\,(\sigma(u))) = \psi\,(\sigma(u))$*, where the first equation follows by* (13) *and* (8)*, and the second follows by* (13) *and* (5)*.*

Hence we can re-express (18) *as*

$$\vdash \mathrm{v} : \psi\,(\sigma(u)),$$

which is the desired result.

$\boxed{\textbf{H-18}}$ *By induction we may assume:*

$$\mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{M}, \tag{1}$$

$$P \cap \mathrm{FV}(\mathcal{C}) = \emptyset, \tag{2}$$

$$P = \{\alpha_0^{\kappa_0}, \ldots, \alpha_{n-1}^{\kappa_{n-1}}\}, \tag{3}$$

$$\begin{aligned}
&\exists\varphi.\mathrm{Dom}(\varphi) = Q \;\wedge \\
&\quad \forall\psi,\sigma,\mathcal{E},\mathrm{V}. \\
&\qquad \vdash \mathcal{E} : \psi\,(\sigma(\mathcal{C}[\mathrm{X} : \mathcal{M}])) \supset \\
&\qquad\quad \mathcal{E} \vdash \mathrm{m} \downarrow \mathrm{V} \supset \\
&\qquad\qquad \vdash \mathrm{V} : \psi\,(\sigma(\varphi\,(\mathcal{M}'))),
\end{aligned} \tag{4}$$

$$Q' \cap (P \cup \mathrm{FV}(\mathcal{M}) \cup \mathrm{FV}(\exists Q.\mathcal{M}')) = \emptyset, \tag{5}$$

$$[Q'/Q] = \{\beta^{\kappa} \mapsto \beta^{\kappa_0 \to \cdots \kappa_{n-1} \to \kappa}\ \alpha_0 \cdots \alpha_{n-1} | \beta^{\kappa} \in Q\}, \tag{6}$$

$$Q' = \{\beta^{\kappa_0 \to \cdots \kappa_{n-1} \to \kappa} | \beta^{\kappa} \in Q\}. \tag{7}$$

We need to show:

$$\begin{aligned}
&\exists\varphi'.\mathrm{Dom}(\varphi') = Q' \;\wedge \\
&\quad \forall\psi,\sigma,\mathcal{E},\mathrm{V}. \\
&\qquad \vdash \mathcal{E} : \psi\,(\sigma(\mathcal{C})) \supset \\
&\qquad\quad \mathcal{E} \vdash \textbf{functor}(\mathrm{X} : \mathrm{S})\mathrm{m} \downarrow \mathrm{V} \supset \\
&\qquad\qquad \vdash \mathrm{V} : \psi\,(\sigma(\varphi'\,(\forall P.\mathcal{M} \to [Q'/Q]\,(\mathcal{M}')))).
\end{aligned}$$

By induction hypothesis (4) *there is some* φ *such that:*

$$\mathrm{Dom}(\varphi) = Q, \tag{8}$$

$$\begin{array}{l} \forall \psi, \sigma, \mathcal{E}, \mathrm{V}. \\ \quad \vdash \mathcal{E} : \psi\,(\sigma(\mathcal{C}[\mathrm{X} : \mathcal{M}])) \supset \\ \qquad \mathcal{E} \vdash \mathrm{m} \downarrow \mathrm{V} \supset \\ \qquad\quad \vdash \mathrm{V} : \psi\,(\sigma(\varphi\,(\mathcal{M}'))). \end{array} \tag{9}$$

Choose

$$\varphi' = \{\beta^{\kappa_0 \to \cdots \kappa_{n-1} \to \kappa} \mapsto \Lambda\alpha_0^{\kappa_0}, \ldots, \alpha_{n-1}^{\kappa_{n-1}}.\varphi\,(\beta^{\kappa}) \mid \beta^{\kappa} \in Q\}.$$

Clearly, by (7), *we have:*

$$\mathrm{Dom}(\varphi') = Q'. \tag{10}$$

Moreover, by (3), (5) *and our choice of* φ'*:*

$$P \cap \mathrm{Inv}(\varphi') = \emptyset. \tag{11}$$

Consider arbitrary ψ, σ, $\mathcal{E}$, V *such that:*

$$\vdash \mathcal{E} : \psi\,(\sigma(\mathcal{C})), \tag{12}$$

$$\mathcal{E} \vdash \mathbf{functor}(\mathrm{X} : \mathrm{S})\mathrm{m} \downarrow \mathrm{V}. \tag{13}$$

W.l.o.g. we can assume:

$$P \cap (\mathrm{Inv}(\psi) \cup \mathrm{FV}(\sigma)) = \emptyset. \tag{14}$$

We need to show:

$$\vdash \mathrm{V} : \psi\left(\sigma(\varphi'\left(\forall P.\mathcal{M} \to [Q'/Q]\left(\mathcal{M}'\right)\right))\right).$$

By assumption (14), *our choice of* φ', (11) *and* (5), *this can be re-expressed as:*

$$\vdash \mathrm{V} : \forall P.\psi\,(\sigma(\mathcal{M})) \to \psi\left(\sigma(\varphi\left(\mathcal{M}'\right))\right).$$

Inverting (13) *by the evaluation rule for functors we must have:*

$$\mathrm{V} \equiv <\mathrm{X}, \mathcal{E}, \mathrm{m}>. \tag{15}$$

Thus it suffices to show:

$$\vdash <\mathrm{X}, \mathcal{E}, \mathrm{m}> : \forall P . \psi\left(\sigma(\mathcal{M})\right) \rightarrow \psi\left(\sigma(\varphi\left(\mathcal{M}'\right))\right).$$

Inverting this goal by the classification rule for functor closures it suffices to show:

$$\begin{array}{l} \forall \delta . \mathrm{Dom}(\delta) = P \supset \\ \quad \forall \mathrm{V}' . \vdash \mathrm{V}' : \delta\left(\psi\left(\sigma(\mathcal{M})\right)\right) \supset \\ \quad\quad \forall \mathrm{V}'' . \mathcal{E}[\mathrm{X} = \mathrm{V}'] \vdash \mathrm{m} \downarrow \mathrm{V}'' \supset \\ \quad\quad\quad \vdash \mathrm{V}'' : \delta\left(\psi\left(\sigma(\varphi\left(\mathcal{M}'\right))\right)\right). \end{array}$$

Consider an arbitrary δ *such that:*

$$\mathrm{Dom}(\delta) = P. \tag{16}$$

Consider an arbitrary V' *such that:*

$$\vdash \mathrm{V}' : \delta\left(\psi\left(\sigma(\mathcal{M})\right)\right). \tag{17}$$

Consider an arbitrary V'' *such that:*

$$\mathcal{E}[\mathrm{X} = \mathrm{V}'] \vdash \mathrm{m} \downarrow \mathrm{V}''. \tag{18}$$

It remains to show:

$$\vdash \mathrm{V}'' : \delta\left(\psi\left(\sigma(\varphi\left(\mathcal{M}'\right))\right)\right).$$

Let $\psi' = \psi \mid \delta$. *Then, by* (2), (14) *and* (16), *we have:*

$$\psi'\left(\sigma(\mathcal{C})\right) = \psi\left(\sigma(\mathcal{C})\right). \tag{19}$$

Moreover, by (14) *and* (16), *we also have:*

$$\psi'\left(\sigma(\mathcal{M})\right) = \delta\left(\psi\left(\sigma(\mathcal{M})\right)\right). \tag{20}$$

Combining (12) *and* (17) *we can show:*

$$\vdash \mathcal{E}[\mathrm{X} = \mathrm{V}'] : \left(\psi\left(\sigma(\mathcal{C})\right)\right)[\mathrm{X} : \delta\left(\psi\left(\sigma(\mathcal{M})\right)\right)],$$

which, by (19) *and* (20), *may be expressed as:*

$$\vdash \mathcal{E}[\mathrm{X} = \mathrm{V}'] : \psi'\left(\sigma(\mathcal{C}[\mathrm{X} : \mathcal{M}])\right). \tag{21}$$

By (9) *(the second half of the induction hypothesis) applied to* ψ', σ, $\mathcal{E}[\mathrm{X} = \mathrm{V}']$, V'', (21) *and* (18) *we obtain:*

$$\vdash \mathrm{V}'' : \psi' \left(\sigma(\varphi \left(\mathcal{M}'\right))\right), \tag{22}$$

Now by assumption (11), (22) *may be re-expressed as:*

$$\vdash \mathrm{V}'' : \delta \left(\psi \left(\sigma(\varphi \left(\mathcal{M}'\right))\right)\right).$$

which is the desired result.

We can now consider the effect that adding the phrase **unpack** e **as** S has on our proof of Property 7.3. The counter-example in Figure 7.12 already demonstrates that the proof must fail because the type soundness property does not hold, but it is revealing to see where it goes wrong. The obvious evaluation rule for the phrase **unpack** e **as** S is:

$$\frac{\mathcal{E} \vdash \mathrm{e} \downarrow <\mathrm{V}>}{\mathcal{E} \vdash \textbf{unpack}\ \mathrm{e}\ \textbf{as}\ \mathrm{S} \downarrow \mathrm{V}.}$$

Unfortunately, the addition of the phrase means that we can no longer prove the stronger property on module expressions used in the proof of Property 7.3. In particular, we can no longer establish the induction hypothesis needed to argue that the classification rule for applicative functors (Rule (H-18)) is sound.

It is easy to see why the proof breaks down. Let's attempt to prove the new case corresponding to the classification rule of **unpack** e **as** S (Rule ($\star$)). According to this rule, the induction hypothesis on e tells us that

$$\forall \psi, \sigma, \mathcal{E}, \mathrm{v}. \vdash \mathcal{E} : \psi \left(\sigma(\mathcal{C})\right) \supset\ \mathcal{E} \vdash \mathrm{e} \downarrow \mathrm{v} \supset\ \vdash \mathrm{v} : \psi \left(\sigma(<\exists P.\mathcal{M}>)\right),$$

However, because **unpack** e **as** S is a *module expression*, we actually need to prove that there is some φ such that:

$$\mathrm{Dom}(\varphi) = P,$$

$$\begin{array}{l} \forall \psi, \sigma, \mathcal{E}, \mathrm{V}. \\ \quad \vdash \mathcal{E} : \psi \left(\sigma(\mathcal{C})\right) \supset \\ \qquad \mathcal{E} \vdash \textbf{unpack}\ \mathrm{e}\ \textbf{as}\ \mathrm{S} \downarrow \mathrm{V} \supset \\ \qquad\quad \vdash \mathrm{V} : \psi \left(\sigma(\varphi \left(\mathcal{M}\right))\right). \end{array}$$

Notice that the realisation φ must be static: since it must hold for all dynamic environments $\mathcal{E}$, it cannot vary with the interpretation of $\mathcal{E}$. Since there is no obvious candidate for φ, let's delay making the choice of φ to see if the induction hypothesis is of any help. Consider an arbitrary ψ, σ, $\mathcal{E}$, and V such that $\vdash \mathcal{E} : \psi\,(\sigma(\mathcal{C}))$ and $\mathcal{E} \vdash$ **unpack** e **as** S $\downarrow$ V. W.l.o.g. we can assume $P \cap (\mathrm{Inv}(\psi) \cup \mathrm{FV}(\sigma)) = \emptyset$. Then, inverting the evaluation rule for **unpack** e **as** S, we know that $\mathcal{E} \vdash$ e $\downarrow$ $<\mathrm{V}>$. The induction hypothesis applied to ψ, σ, $\mathcal{E}$, and $<\mathrm{V}>$ tells us that $\vdash$ $<\mathrm{V}>$: $\psi\,(\sigma(<\exists P.\mathcal{M}>))$, which, by our assumption on P, is equivalent to $\vdash$ $<\mathrm{V}>$: $<\exists P.\psi\,(\sigma(\mathcal{M}))>$. Inverting the classification rule for encapsulated module values we can establish that there is some realisation φ' with $\mathrm{Dom}(\varphi') = P$ such that $\vdash$ V : $\varphi'\,(\psi\,(\sigma(\mathcal{M})))$. Unfortunately, this realisation φ' is of no use in determining the static realisation φ because it depends on the dynamic environment $\mathcal{E}$. In short, adding the phrase **unpack** e **as** S violates the invariant required to support applicative functors.

7.5 Conclusion

The addition of first-class modules extends the expressive power of both Core-ML and Higher-Order Modules considerably. Although seductively easy to define by the addition of a new form of simple type and corresponding denotation, introduction and elimination rules, we need to tread carefully. In particular, the equivalence of semantic simple types and the definition of realisation and enrichment are now mutually dependent. Therefore, we should not immediately presume that the algorithm for solving matching problems remains well-behaved. We shall not investigate the meta-theory of First-Class Modules any further in this thesis, leaving it to future work. We will, however, touch upon First-Class Modules again briefly at the end of Chapter 8. There, we will consider the practical problem of performing Core-ML type inference in the presence of First-Class Modules and propose a tentative solution.

Chapter 8

Type Inference for Core-ML with Higher-Order and First-Class Modules

The aim of this chapter is to design an algorithm that integrates Core-ML type inference with Modules type checking. The primary motivation for this work is that it paves the way for the correct integration of our proposals with existing implementations of Standard ML. **Although we take care to present our algorithms with their intended correctness properties, the *verification* of these properties is left to future work.**

In previous chapters, we focused on Modules and took the Core language for granted. In Chapter 5, we laid the foundations for type-checking Higher-Order Modules by presenting a signature matching algorithm that terminates and is sound and complete for the matching problems encountered during module classification (Sections 5.6 and 5.7). Using these results, it is straightforward to derive a *type checking algorithm* for Higher-Order Modules directly from the rules of the static semantics by replacing premises requiring the existence of matching realisations by appeals to the signature matching algorithm. The resulting algorithm has the following behaviour: given as input a context $\mathcal{C}$ and a phrase p, the algorithm terminates with one of two results: it either *succeeds* with a semantic object o as its output or it *fails*. The algorithm is *sound* in the sense that if it succeeds with o, and p is a type phrase, then the denotation judgement $\mathcal{C} \vdash \mathrm{p} \triangleright o$ holds; if p is a term phrase, then the classification judgement $\mathcal{C} \vdash \mathrm{p} : o$ holds. The algorithm is *complete* in the sense that, if it does fail, then this is because there is no object o to which the phrase may be related: if p is a type phrase, it fails to denote; if p is a term phrase, it fails to be well-typed.

The soundness and completeness of this algorithm is predicated on the existence of analogous type checkers for type and term phrases of the Core. As an hypothesis concerning an arbitrary Core language, this simplifying assumption is reasonable: a wide variety of strongly typed Core languages admit type checking algorithms of this kind. Unfortunately, as we shall see, Core-ML does not.

Core-ML, too, is a strongly typed language. But it differs from most other such languages by being *implicitly* typed. As in ordinary ML, a function λi.e does not declare the type of its parameter i. Similarly, an occurrence of a polymorphic phrase vo gives no indication of the monomorphic type at which it is used. In ML, the justification for omitting the type of a λ-bound identifier is that it can be recovered in a principled manner by examining both the ways in which the identifier is used within the body of the function and the types of the arguments to which the function is applied. Similarly, the monomorphic instance of a polymorphic phrase can be determined from both the phrase's polymorphic type, and the way in which this phrase is used. For ML, the magic that makes this possible is Milner's well known *type inference* algorithm, algorithm $\mathcal{W}$ (and its many variants). Algorithm $\mathcal{W}$ satisfies its own form of soundness and completeness theorems with respect to the static semantics of ML.

Given the similarities between Core-ML and ordinary ML, let us assume for the moment that we can adapt $\mathcal{W}$ to provide type inference for Core-ML. Unfortunately, we cannot just naively plug it into the type checker for Higher-Order Modules to obtain a combined type inference and type checking algorithm for the entire language. The problem is that $\mathcal{W}$ does not belong to the family of generic Core type checkers from which we can construct the type checker for Higher-Order Modules. It is easy to see this by comparing the output and soundness properties that we assumed of the generic Core's type checker with the actual output and soundness property of $\mathcal{W}$. For the generic Core, we assumed that the input to the type checker is a context $\mathcal{C}$ and a value expression e and that the successful output is a value type v such that $\mathcal{C} \vdash \mathrm{e} : v$. The input to $\mathcal{W}$ is also a context $\mathcal{C}$ and a value expression e. However, the successful output of $\mathcal{W}$ is not just a type but a pair, consisting of both a type v and an inferred *substitution* σ, mapping simple type variables to simple types. The soundness property of $\mathcal{W}$ tells us not that $\mathcal{C} \vdash \mathrm{e} : v$ but rather that $\sigma(\mathcal{C}) \vdash \mathrm{e} : v$. The substitution is the necessary by-product of performing type *inference* instead of mere type checking. It captures the minimal type information that needs to be inferred about the original context to make the expression well-typed: $\sigma(\mathcal{C})$ is the *inferred* context. Thus we encounter the first difficulty posed by integrating Core-ML type inference with Modules type checking: the Core-ML type inference algorithm does not meet the requirements of the Modules type checking algorithm.

Unfortunately, the converse is also true: our type checking algorithm for Modules does not satisfy the requirements of an $\mathcal{W}$-style type inference algorithm for Core-ML. Core-ML, because it must cater for Modules, extends the grammar of ordinary ML expressions with the phrase vo, where vo $\in$ ValOcc is a value occurrence. Value occurrences provide access to values defined in the context and within structures. In Higher-Order Modules, the inclusion of value occurrences of the form m.x means that a Core-ML expression can contain an arbitrary module expression as a subphrase. This is also true in First-Class Modules, since it extends the grammar of Core-ML expression with the phrase **pack** m **as** S, where m is a module expression. This has considerable ramifications for both Core-ML type inference and Modules type checking.

For instance, consider the type inference problem posed by a Core-ML function of the form $\lambda\mathrm{i.e}[\mathrm{m}[\mathrm{i}], \mathrm{i}]$[1]. For soundness, the type inferred for the function parameter i must be consistent with every type at which i is used within the function's body $\mathrm{e}[\mathrm{m}[\mathrm{i}], \mathrm{i}]$. In particular, it must also be consistent with each free occurrence of i in the enclosed module expression m[i]. It should be clear that the algorithm for type checking the module expression m[i] must be able to contribute to the result of the algorithm inferring the type of the Core-ML function. Moreover, since the type of i may not be fully determined before the inspection of m[i], the Modules type checker must be able to proceed with only imperfect knowledge of the context. The type checking algorithm of Chapter 5 falls short of these requirements and we shall have to adapt it to support Core-ML type inference.

Finally, the static semantics of Modules places an additional requirement on type inference that goes beyond those dictated by the static semantics of ML: the Core-ML type inference algorithm must respect any side-conditions on type variables imposed by the rules of the static semantics. A specific example is the functor introduction rule that has a side-condition stipulating that the functor's type parameters do not occur free in the context. For instance, to respect this side-condition when inferring the type of a Core-ML function of the form λi.e[**functor**(X : **sig type** t : k **end**)m[i]], the algorithm must ensure that the denotation of the type parameter X.t cannot appear

[1]The notation $\mathrm{p}[\mathrm{p}_1, \ldots, \mathrm{p}_n]$ signifies that the subphrases $\mathrm{p}_1, \ldots, \mathrm{p}_n$ have some non-overlapping occurrences in the phrase p.

in the type inferred for i. Since the static semantics of ML does not impose such side-conditions, algorithm $\mathcal{W}$ was not designed to respect them. To adapt $\mathcal{W}$ to Core-ML requires a non-trivial extension of both $\mathcal{W}$ and its underlying unification algorithm.

Section 8.1 provides some background: we review ML and its static semantics, define substitution and unification and use these concepts to present algorithm $\mathcal{W}$. Section 8.2 gives examples illustrating the issues that need to be addressed when designing a type inference algorithm for Core-ML in the presence of Higher-Order Modules. In Section 8.3 we design new unification and matching algorithms that can solve the more difficult problems encountered in this setting, and then use these algorithms to design a type inference algorithm for Core-ML with Higher-Order Modules. The algorithm is similar in spirit to $\mathcal{W}$. In Section 8.4 we briefly suggest how to "tie the knot" to obtain a type inference algorithm for First-Class Modules (Core-ML extended with package types). The existence of this algorithm makes the proposals of Chapter 7 a little more concrete.

Section 8.5 closes with a brief assessment.

8.1 A Review of ML Type Inference

Figure 8.1 presents the grammar and semantic objects of ordinary ML, the language originally proposed and studied by Milner [Mil78].

A brief comparison of ML with Core-ML is appropriate. The ML phrases **let** x = e **in** e′ and x define and eliminate identifiers with polymorphic types. Although easily incorporated, we omitted these phrases from Core-ML since their roles are similar to the ones played by value definitions and value occurrences. Also note that, because ML is implicitly typed, there is no need for type phrases. Core-ML provides type phrases solely for the benefit of Modules, where they are required in order to define and specify structure components. Turning to the semantic objects, observe that ML's simple types $u \in \mathit{SimTyp}$ are less complicated than their Core-ML counterparts. Indeed, in ML the equivalence between simple types is purely syntactic. In Core-ML, on the other hand, a simple type may consist of an application $\nu(u_0, \ldots, u_{\mathrm{k}-1}) \in \mathit{SimTyp}$ of a type name $\nu \in \mathit{TypNam}$ and we need to identify applications that are equivalent "up to" the equivalence of type names.

Figure 8.2 defines the static semantics of ML. Our description of ML's static semantics and algorithm $\mathcal{W}$ is closer to Tofte's equivalent, but more modern, presentation of Milner's work [Tof88].

$$
\begin{array}{rcll}
\mathrm{e} & ::= & \mathrm{i} & \text{monomorphic identifier} \\
 & | & \lambda \mathrm{i}.\mathrm{e} & \lambda\text{-abstraction} \\
 & | & \mathrm{e}\ \mathrm{e}' & \text{application} \\
 & | & \mathrm{x} & \textit{polymorphic identifier} \\
 & | & \textbf{let}\ \mathrm{x} = \mathrm{e}\ \textbf{in}\ \mathrm{e}' & \textit{polymorphic definition}
\end{array}
$$

(a) Grammar

$$
\begin{array}{lcll}
u \in \mathit{SimTyp} & ::= & {}'a \mid u \rightarrow u' & \\
v \in \mathit{ValTyp} & ::= & \forall{}'a_0, \ldots, {}'a_{n-1}.u & \text{polymorphic simple type}
\end{array}
$$

$$
\mathcal{C} \in \mathit{Context} \quad \stackrel{\mathrm{def}}{=} \quad \left\{ \mathcal{C}_{\mathrm{i}} \cup \mathcal{C}_{\mathrm{x}} \;\middle|\; \begin{array}{l} \mathcal{C}_{\mathrm{i}} \in \mathrm{CoreId} \xrightarrow{\mathrm{fin}} \mathit{SimTyp}, \\ \mathcal{C}_{\mathrm{x}} \in \mathrm{ValId} \xrightarrow{\mathrm{fin}} \mathit{ValTyp} \end{array} \right\}
$$

(b) Semantic Objects

Figure 8.1: ML's Grammar and Semantic Objects

The classification judgement $\mathcal{C} \vdash \mathrm{e} : u$ relates the expression e to a *monomorphic* simple type u that it inhabits. The separate classification judgement $\mathcal{C} \vdash \mathrm{e} : v$, on the other hand, relates e to a value type v, i.e. a quantified simple type. Rule (*ML*-4) eliminates the polymorphism of an identifier. The rule employs a generalisation relation $\_ \succ \_ \in \mathit{ValTyp} \times \mathit{SimTyp}$ that is defined as in Definition 3.27. Rule (*ML*-5) introduces a polymorphic identifier by deriving a polymorphic type for its definition. Rule (*ML*-6) derives a value type of an expression from a simple type of that expression, by universally quantifying over every type variable that occurs free in the simple type, without occurring free in the context. Notice that none of the rules impose side-conditions on type variables, e.g. premises of the form ${}'a \notin \mathrm{FTVS}(\mathcal{C})$, but that similar conditions are ubiquitous in the static semantics of Core-ML and Modules.

Observe that an expression may inhabit more than one simple type. For

Monomorphic Values $\boxed{\mathcal{C} \vdash \mathrm{e} : u}$

$$\frac{\mathrm{i} \in \mathrm{Dom}(\mathcal{C}) \quad \mathcal{C}(\mathrm{i}) = u}{\mathcal{C} \vdash \mathrm{i} : u} \qquad (\mathit{ML}\text{-}1)$$

$$\frac{\mathcal{C}[\mathrm{i} : u] \vdash \mathrm{e} : u'}{\mathcal{C} \vdash \lambda \mathrm{i}.\mathrm{e} : u \to u'} \qquad (\mathit{ML}\text{-}2)$$

$$\frac{\mathcal{C} \vdash \mathrm{e} : u' \to u \quad \mathcal{C} \vdash \mathrm{e}' : u'}{\mathcal{C} \vdash \mathrm{e}\,\mathrm{e}' : u} \qquad (\mathit{ML}\text{-}3)$$

$$\frac{\mathrm{x} \in \mathrm{Dom}(\mathcal{C}) \quad \mathcal{C}(\mathrm{x}) = v \quad v \succ u}{\mathcal{C} \vdash \mathrm{x} : u} \qquad (\mathit{ML}\text{-}4)$$

$$\frac{\mathcal{C} \vdash \mathrm{e} : v \quad \mathcal{C}[\mathrm{x} : v] \vdash \mathrm{e}' : u}{\mathcal{C} \vdash \mathbf{let}\ \mathrm{x} = \mathrm{e}\ \mathbf{in}\ \mathrm{e}' : u} \qquad (\mathit{ML}\text{-}5)$$

Polymorphic Values $\boxed{\mathcal{C} \vdash \mathrm{e} : v}$

$$\frac{\mathcal{C} \vdash \mathrm{e} : u \quad \{{}'a_0, \ldots, {}'a_{n-1}\} = \mathrm{FTVS}(u) \setminus \mathrm{FTVS}(\mathcal{C})}{\mathcal{C} \vdash \mathrm{e} : \forall {}'a_0, \ldots, {}'a_{n-1}.u} \qquad (\mathit{ML}\text{-}6)$$

Figure 8.2: Static Semantics of ML.

instance, the identity function λi.i has types:

$$\begin{gathered} 'a \to 'a, \\ ('a \to 'a) \to ('a \to 'a), \\ ('a \to 'b) \to ('a \to 'b), \\ \vdots \end{gathered}$$

In fact, it can be assigned the type $u \to u$, for any simple type u. By using universal quantification over type variables, the static semantics of ML can internalise this observation, assigning the identity the *polymorphic* value type $\forall 'a.'a \to 'a$. This particular value type is obtained by quantifying over $'a$ in the identity's simple type $'a \to 'a$. Notice, however, that the rules equally allow us to assign the less general value type $\forall 'a.('a \to 'a) \to ('a \to 'a)$, by quantifying over $'a$ in the identity's more specific simple type $('a \to 'a) \to ('a \to 'a)$.

Fortunately, among the value types that can be assigned to expressions, there are some that are most general, according to a pre-order $\_ \succeq \_ \in$ *ValTyp* $\times$ *ValTyp* on ML value types that is defined just as in Definition 3.28 of Core-ML:

Definition 8.1 (Principal Value Types).
A value type v is *principal* for e in $\mathcal{C}$ if, and only if, $\mathcal{C} \vdash$ e : v and $v \succeq v'$ whenever $\mathcal{C} \vdash$ e : v'.

It can be shown that whenever an expression has a simple type in a given context, then it also has a principal value type in that context.

It should be clear that the rules in Figure 8.2 do not describe an algorithm for determining the principal type of an expression. The rule for typing an abstraction λi.e is non-deterministic since it does not specify which simple type u to assume for i. The rule for using a polymorphic value x of type v does not uniquely determine which particular instance u of v to return. Even the **let**-rule is non-deterministic, since it fails to indicate which value type to assume for x.

This is where Milner's algorithm $\mathcal{W}$ comes in. Given a context $\mathcal{C}$ and an expression e, $\mathcal{W}$ returns the principal value type of e in a context derived from $\mathcal{C}$. The definition of $\mathcal{W}$, which we shall shortly present, relies on the more primitive concepts of *substitution* and *unification*.

Definition 8.2 (Substitution). A *substitution* σ is a finite map from simple type variables to (semantic) simple types:

$$\sigma \in \mathit{Subst} = \mathit{SimTypVar} \overset{\text{fin}}{\rightarrow} \mathit{SimTyp}.$$

The operation of *applying* a substitution σ to a semantic object o, written $\sigma(o)$, is defined in the usual way, taking care to avoid capture of free variables by binding constructs.

The *composition* of two substitutions, written $\sigma_2 \circ \sigma_1$, is defined as the essentially unique substitution, satisfying,

$$(\sigma_2 \circ \sigma_1)(o) = \sigma_2(\sigma_1(o)),$$

for every semantic object o.

Given two substitutions σ and σ_1, σ is *more general* than σ_1 written $\sigma \succeq \sigma_1$, if, and only if:

$$\exists \sigma_2.\ \sigma_2 \circ \sigma = \sigma_1.$$

The substitution $\sigma \setminus R$, where R is a set of simple type variables, is defined as the substitution that is equivalent to σ on the restricted domain $\mathrm{Dom}(\sigma) \setminus R$, i.e. $\sigma \setminus R \overset{\text{def}}{=} \{'a \mapsto \sigma('a) | 'a \in \mathrm{Dom}(\sigma) \setminus R\}$.

During its execution, $\mathcal{W}$ sets up tentative equations between pairs of simple types containing type variables. Such equations must be solved by finding substitutions that make them hold. Formally, we define:

Definition 8.3 (Unification Problems and Most General Unifiers).

- A *unification problem* is a pair of simple types u and u'.
- A *unifier* of u and u' is a substitution σ such that $\sigma(u) = \sigma(u')$. A unifier of u and u' is a *solution* to the unification problem posed by u and u'.
- A *most general unifier* of u and u' is a unifier σ of u and u' such that, for every other unifier σ_1 of u and u', $\sigma \succeq \sigma_1$. A most general unifier of u and u' is a *principal solution* to the unification problem posed by u and u'.

A particular unification problem may fail to have a unifier. However, if it does have a unifier, then it also has a most general unifier. Figure 8.3 shows a simple implementation of a deterministic algorithm that solves unification problems, returning a most general unifier whenever one exists, and failing otherwise. The side condition $'a \notin \mathrm{FTVS}(u)$ on Rules ($\mathcal{R}$-2) and ($\mathcal{R}$-3) is

Unification of Simple Types $\boxed{\vdash u = u' \downarrow \sigma}$

$$\frac{}{\vdash {}'a = {}'a \;\downarrow\; \emptyset} \tag{$\mathcal{R}$-1}$$

$$\frac{{}'a \notin \mathrm{FTVS}(u)}{\vdash {}'a = u \;\downarrow\; [u/{}'a]} \tag{$\mathcal{R}$-2}$$

$$\frac{{}'a \notin \mathrm{FTVS}(u) \quad \forall{}'b \in \mathit{SimTypVar}.u \not\equiv {}'b}{\vdash u = {}'a \;\downarrow\; [u/{}'a]} \tag{$\mathcal{R}$-3}$$

$$\frac{\vdash u_1 = u_1' \;\downarrow\; \sigma_1 \quad \vdash \sigma_1(u_2) = \sigma_1(u_2') \;\downarrow\; \sigma_2}{\vdash u_1 \to u_2 = u_1' \to u_2' \;\downarrow\; \sigma_2 \circ \sigma_1} \tag{$\mathcal{R}$-4}$$

Figure 8.3: Robinson's Unification Algorithm.

known as the *occur check* and is used to rule out substitutions that assume the existence of infinite types. The side condition $\forall{}'b \in \mathit{SimTypVar}.u \not\equiv {}'b$ on Rule ($\mathcal{R}$-3) ensures that only Rule ($\mathcal{R}$-2) applies in the special case that both types are distinct variables. This implementation is based on Robinson's original unification procedure [Rob65]. Its correctness is captured by the following theorem:

Theorem 8.4 (Correctness of Unification). *[Rob65]*
For any simple types u and u':

Termination *The appeal $\vdash u = u' \downarrow \_$ terminates either in success, returning a substitution, or failure.*

Soundness *If $\vdash u = u' \downarrow \sigma$ then $\sigma(u) = \sigma(u')$.*

Completeness *If $\sigma_1(u) = \sigma_1(u')$ then, for some σ, $\vdash u = u' \downarrow \sigma$ with $\sigma \succeq \sigma_1$.*

Figure 8.4 presents Milner's algorithm $\mathcal{W}$ as a collection of inference rules. The algorithm takes a context and an expression as input, and outputs a pair of a value type and substitution on type variables. At the heart of

Monomorphic Values $\boxed{\mathcal{C} \vdash \mathrm{e} \downarrow u, \sigma}$

$$\frac{\mathrm{i} \in \mathrm{Dom}(\mathcal{C}) \quad \mathcal{C}(\mathrm{i}) = u}{\mathcal{C} \vdash \mathrm{i} \downarrow u, \emptyset} \qquad (\mathcal{W}\text{-}1)$$

$$\frac{{}'a\ \textbf{fresh} \quad \mathcal{C}[\mathrm{i} : {}'a] \vdash \mathrm{e} \downarrow u', \sigma}{\mathcal{C} \vdash \lambda \mathrm{i}.\mathrm{e} \downarrow \sigma({}'a) \to u', \sigma \setminus \{{}'a\}} \qquad (\mathcal{W}\text{-}2)$$

$$\frac{\begin{array}{l}\mathcal{C} \vdash \mathrm{e} \downarrow u, \sigma_1 \\ \sigma_1(\mathcal{C}) \vdash \mathrm{e}' \downarrow u', \sigma_2 \\ {}'a\ \textbf{fresh} \\ \vdash \sigma_2(u) = u' \to {}'a \ \downarrow \ \sigma_3\end{array}}{\mathcal{C} \vdash \mathrm{e}\ \mathrm{e}' \downarrow \sigma_3({}'a), (\sigma_3 \circ \sigma_2 \circ \sigma_1) \setminus \{{}'a\}} \qquad (\mathcal{W}\text{-}3)$$

$$\frac{\mathrm{x} \in \mathrm{Dom}(\mathcal{C}) \quad \mathcal{C}(\mathrm{x}) = \forall {}'a_0, \ldots, {}'a_{n-1}.u \quad \forall i \in [n].{}'a_i\ \textbf{fresh}}{\mathcal{C} \vdash \mathrm{x} \downarrow u, \emptyset} \qquad (\mathcal{W}\text{-}4)$$

$$\frac{\mathcal{C} \vdash \mathrm{e} \downarrow v, \sigma_1 \quad \sigma_1(\mathcal{C})[\mathrm{x} : v] \vdash \mathrm{e}' \downarrow u, \sigma_2}{\mathcal{C} \vdash \textbf{let}\ \mathrm{x} = \mathrm{e}\ \textbf{in}\ \mathrm{e}' \downarrow u, \sigma_2 \circ \sigma_1} \qquad (\mathcal{W}\text{-}5)$$

Polymorphic Values $\boxed{\mathcal{C} \vdash \mathrm{e} \downarrow v, \sigma}$

$$\frac{\begin{array}{l}\mathcal{C} \vdash \mathrm{e} \downarrow u, \sigma \\ \{{}'a_0, \ldots, {}'a_{n-1}\} = \mathrm{FTVS}(u) \setminus \mathrm{FTVS}(\sigma(\mathcal{C}))\end{array}}{\mathcal{C} \vdash \mathrm{e} \downarrow \forall {}'a_0, \ldots, {}'a_{n-1}.u, \sigma} \qquad (\mathcal{W}\text{-}6)$$

Figure 8.4: Milner's algorithm $\mathcal{W}$.

algorithm $\mathcal{W}$ lies an appeal to unification (Rule ($\mathcal{W}$-3)). Rules ($\mathcal{W}$-2), ($\mathcal{W}$-4) and ($\mathcal{W}$-3) employ side conditions, e.g. $'a$ **fresh**, stipulating the choice of a new variable drawn from some infinite supply of currently *unused* variables. For the moment, we shall follow Milner and Tofte [Mil78, Tof88] and be vague about what it means to choose a "fresh" variable. We shall be more precise about this in our own algorithms.

Algorithm $\mathcal{W}$ satisfies the following theorem:

Theorem 8.5 (Correctness of $\mathcal{W}$). *[Mil78, Dam85]*
For every context $\mathcal{C}$ and expression e*:*

Termination *The appeal $\mathcal{C} \vdash \mathrm{e} \downarrow \_, \_$ terminates either in success, returning a value type and a substitution, or failure.*

Soundness *If $\mathcal{C} \vdash \mathrm{e} \downarrow v, \sigma$, then $\sigma(\mathcal{C}) \vdash \mathrm{e} : v$.*

Completeness *If $\sigma_1(\mathcal{C}) \vdash \mathrm{e} : v_1$ then for some v, σ and σ_2, $\mathcal{C} \vdash \mathrm{e} \downarrow v, \sigma$ with $\sigma_2 \circ \sigma = \sigma_1$ and $\sigma_2(v) \succeq v_1$.*

Indeed, the termination, soundness and completeness properties of $\mathcal{W}$ are indirect consequences of the corresponding properties of the unification algorithm.

How does $\mathcal{W}$ work? Consider the rules of the static semantics. For each phrase, the type of that phrase, even though it may not be uniquely determined, is at least constrained by three factors: the context, the types of its subphrases, and the form of the phrase itself. For instance, Rule (*ML*-3), classifying an application e e$'$, places an implicit equational constraint on the types of its subphrases: e's type must be equivalent to a function space; moreover, the domain of this function space must be equivalent to the type of the argument e$'$. Algorithm $\mathcal{W}$ proceeds by traversing the structure of the given phrase. If the corresponding rule of the static semantics requires the non-deterministic choice of a suitable type, then $\mathcal{W}$ introduces a "fresh" type variable to represent that choice. The fact that the variable is fresh ensures that the choice is initially unconstrained. If the rule also requires an equation between types to hold, $\mathcal{W}$ attempts to solve the equation by finding a most general substitution for the type variables that makes the equation hold. Since all of the equational constraints between types must hold simultaneously, these substitutions are propagated (i.e. applied to the context) in recursive calls, and accumulated (i.e. composed) in the result.

For a less operational interpretation, we can first observe the following key property of ML and its judgements:

Property 8.6 (Closure under Substitution).

- $v \succ u$ *implies* $\sigma(v) \succ \sigma(u)$.
- $\mathcal{C} \vdash \mathrm{e} : u$ *implies* $\sigma(\mathcal{C}) \vdash \mathrm{e} : \sigma(u)$.
- $\mathcal{C} \vdash \mathrm{e} : v$ *implies* $\sigma(\mathcal{C}) \vdash \mathrm{e} : \sigma(v)$.

Since the classification rules of ML are syntax directed, the algorithm merely attempts to construct the most general typing derivation in the given context, directed by the syntax of the given phrase. At certain points, it uses unification to determine the most general substitution that is necessary in order to apply the corresponding rule from the static semantics. If the phrase involves a recursive call followed by either a second recursive call, or an appeal to unification, both of which return a substitution, then Property 8.6 ensures that the derivation constructed from the first call also gives rise to a valid derivation under this additional substitution.

8.2 Type Inference Issues

Now that we have reviewed type inference for ML, let us consider the main issues that arise when we attempt to adapt algorithm $\mathcal{W}$ to Core-ML with Higher-Order Modules.

The Problem with Interleaving Core-ML Type Inference with Modules Type Checking

At first glance, we might be tempted to think that the tasks of performing type inference for Core-ML and type checking for Modules are orthogonal, i.e. that we can get away with the simple-minded approach of interleaving Core type inference and Modules type checking, alternating between (a variant) of algorithm $\mathcal{W}$ and the type checker for Higher-Order Modules presented in Chapter 5. Such an algorithm would invoke type inference when entering a Core expression, reverting back to type checking when entering a Modules subphrase, and so on. The following examples demonstrate that this approach cannot be made to work.

First, consider this perfectly legal function of type **int** → **int**:

```
        λi.(struct val x = + i 1 end).x
```

Here `i` must be assigned the simple type **int** for the example to type check. Moreover, this type can only be inferred by examining the Core expression `+ i 1` *within* the module expression `struct val x = + i 1 end`. Clearly the type checker responsible for the module expression must feed this information back to the Core type inference algorithm.

Now suppose we adopt the interleaving algorithm. $\mathcal{W}$ proceeds by assigning a fresh type variable $'a$ to `i` and then calls the Modules type checker to calculate the type of the projection `(struct val x = + i 1 end).x`. The type checker descends into the module expression and ultimately calls $\mathcal{W}$ on the expression `+ i 1`. This returns the type **int** and substitution $[\textbf{int}/'a]$. Since the type checker for Modules, by its very design, has no idea what to do with this substitution let's assume it just throws it away. It still knows that `x` has type **int** and returns this as the type of the projection `(struct val x = + i 1 end).x`. So far so good. However, because the substitution was discarded, $\mathcal{W}$ reports the "principal" type of the complete phrase as $\forall 'a.'a \rightarrow \textbf{int}$. Not only is this incorrect, it's not even sound. We can now apply this expression to an argument of arbitrary type: if we apply it to another function, we obtain the sad example of trying to add a function to an integer.

In this example, the approach of ignoring the inferred substitution is flawed because it fails to ensure that the function's domain is consistent with (every) use of its argument. A sound algorithm cannot discard inferred substitutions. Indeed, we will revise our type checking algorithm for module expressions so that it not only checks that phrase are well-typed, but also propagates and accumulates substitutions returned by recursive calls to Core-ML type inference.

Unfortunately, this argument applies to all of the phrase classes of the Modules language. For instance, consider the phrase:

```
λi.(struct type t = (struct type u = int; val x = + i 1 end).u;
           val  y = 1
    end).y
```

Here, the type of `i` is determined by a Core-ML phrase occurring within the *type phrase* `(struct type u = int val x = + i 1 end).u`. Clearly, the algorithm used to determine the denotation of a type phrase must also be modified to support type inference.

```
functor(A:sig type a:0; val a:a end)
   struct val x = λi.((functor(B:sig type b:0; val b:b end)
                         struct val y = λj. if true then i
                                                      else A.a
                         end)
                       struct type b = int; val b = 1 end).y
   end
```

Figure 8.5:

Worse still, consider the phrase:

```
λi.(struct val x = i end \ sig val x:int end).x
```

It is easy to see that it has type **int** $\rightarrow$ **int**. However, the fact that `i` must have type **int** is determined neither by the structure expression `struct val x = i end`, nor by the signature `sig val x:int end`. Instead, it is forced by the requirement that the type of the module `struct val x = i end` *matches* the signature `sig val x:int end`. Thus the algorithm for computing matching realisations between module types must also be modified to support type inference.

Remark 8.2.1. The previous examples all hinge on the use of generalised projections to construct phrases in which a λ-bound Core-ML identifier, whose type must be inferred, appears free in a module expression. It is plausible that any measure taken to enforce the property that all module expression are *closed* with respect to λ-bound identifiers will ensure that it is safe to adopt the interleaving algorithm sketched above. Possibly one of the simplest fixes is to syntactically restrict module projections to *paths* in the sense of Leroy [Ler94, Ler95]. However, this restriction is too strong to be acceptable in First-Class Modules, the extension of Core-ML with package types, so we prefer to tackle the more general problem of inferring types for module phrases containing free λ-bound identifiers directly. Finding a solution to this problem will make it straightforward to adapt type-inference to First-Class Modules.

```
functor(A:sig type a:0; val a:a end)
   struct val x = λi.((functor(B:sig type b:0; val b:b end)
                      struct val y = λj. if true then i
                                                else B.b
                      end)
                    struct type b = int; val b = 1 end).y
   end
```

Figure 8.6:

```
functor(A:sig type a:0; val a:a end)
   struct val x = λi.((functor(B:sig type b:0; val b:b end)
                      struct val y = λj. if true then i
                                                else j
                      end)
                    struct type b = int; val b = 1 end).y
   end
```

Figure 8.7:

```
functor(A:sig type a:0; val a:a end)
  struct val x = λi.((functor(B:sig type b:0; val b:b end)
                     struct val y =
                            λj. pair
                                 (if true then i else j)
                                 (if true then j else B.b)

                     end)
                   struct type b = int; val b = 1 end).y
  end
```

Figure 8.8:

Parameters and Scope

A separate issue concerns the need to ensure that type inference does not violate the side conditions on type variables required by so many of the static semantic rules of Modules.

Consider the type inference problems posed by the (contrived) examples in Figures 8.5, 8.6, 8.7, and 8.8. The first three differ only in the branches of the innermost conditional construct. We shall assume that a conditional expression can be classified only if both of its branches have the same type. Some of these phrases are well-typed, others are not. What is common to all of the phrases is that the body of the outer functor must be parametric in the type `A.a`, and the body of the inner functor must be parametric in both the type `A.a` and the type `B.b`.

Suppose we are trying to construct classifications for these module expressions. Let α be the type parameter denoted by the type `A.a`, and β be the one denoted by the type `B.b`. Let u and u' be the simple types that need to be inferred for `i` and `j`, respectively. To ensure that the variables α and β are really treated as *parameters* representing arbitrary types, the side conditions on the functor introduction rule require that α does not occur in the context of the outer functor, and that β does not occur in the context of the inner functor. In particular, this means that we must ensure the condition $\beta \notin \mathrm{FV}(u)$, since the inner functor is in the scope of the declaration of `i`. On the other hand, α may occur both in u and u', because `i` and `j` are declared within the scope of `A`. Similarly, β may occur in u', since `j` is declared within the scope of `B`. In each example, the difficulty lies in choosing u and u' in a way that makes the conditional expression type-check, without violating the side condition $\beta \notin \mathrm{FV}(u)$.

Example 8.5 can be classified since we can choose $u = \alpha$ and independently choose an arbitrary type for u'.

Example 8.6, on the other hand, cannot be classified since the conditional expression requires $u = \beta$, violating $\beta \notin \mathrm{FV}(u)$. It should be rejected by a sound typing algorithm.

Example 8.7 can be classified provided we choose $u = u'$, for some choice of u' such that $\beta \notin \mathrm{FV}(u')$: since u' is permitted but not required to contain β, we are free to choose any such u'.

Contrast this last example with Example 8.8. Although the first conditional requires $u = u'$, the second also requires that $u' = \beta$; any attempt to satisfy both equations violates the side condition that $\beta \notin \mathrm{FV}(u)$. This phrase cannot be classified, and should be rejected by a sound typing algorithm.

Now suppose that we try to use algorithm $\mathcal{W}$ to infer the types of `i` and `j`. Note that we carried out the above discussion using *linguistic* meta-variables u and u', *describing* generic choices of u and u' as solutions to equations between types. Algorithm $\mathcal{W}$ carries out a similar analysis using *simple type variables* as *syntactic* meta-variables and *computing* most general solutions to equations between types by unification. Unfortunately, we also had to enforce side conditions on parameters but neither algorithm $\mathcal{W}$, nor the underlying unification algorithm, address this issue. In each of the examples, $\mathcal{W}$ would simply assign "fresh" simple type variables $'a$ and $'b$ to `i` and `j`, unify them as required, but fail to check the side condition on β. As a result, it would deem that all of the examples above are well-typed, even though some of them are not: algorithm $\mathcal{W}$ is not sound in this setting.

Note that type variables $\alpha \in \mathit{TypVar}$ are not the only kind of variable that can play the role of parameters in typing derivations. The Core-ML rules (C-4) and (C-5), relating, respectively, definable types and value types to their denotations, both introduce simple type variables into derivations. These variables, too, must be treated as parameters with definite scopes. Moreover, they must never be confused with those simple type variables used as meta-variables during type inference. In particular, they must be prevented from occurring in the domain of any inferred substitution.

To respect the static semantics of Modules we will design a modified unification algorithm that keeps track of the relative scopes of parameters and meta-variables, using this information to ensure that any side-conditions on parameters are never violated by unification. In turn, we will have to design our Modules type checker and Core-ML type inference algorithm to perform some additional bookkeeping, beyond that undertaken by algorithm $\mathcal{W}$: to set the scene for any appeals to unification, each algorithm will need to record the scope and role of every type variable that is introduced during its execution.

8.3 Type Inference for Core-ML with Higher-Order Modules

In this section, we shall design a type inference algorithm for Core-ML with Higher-Order Modules that addresses the issues identified in Section 8.2.

8.3.1 Unification of Core-ML Simple Types

At a more abstract level, a class of unification problems is determined by a set of terms supporting both a notion of substitution and an underlying equivalence on terms. For a given class of problems, an instance of the unification problem is a pair of terms drawn from the set of terms. The unification problem is to construct a (preferably principal) substitution that equates the terms according to the underlying equivalence, provided such a substitution exists.

The class of unification problems encountered in algorithm $\mathcal{W}$ is particularly simple:

- The terms to be unified are simple types constructed from a reduced grammar of first-order type variables and the function space between types.
- The equivalence between simple types is purely syntactic.
- Every simple type variable occurring in a unification problem posed by $\mathcal{W}$ is in fact a meta-variable that is game for substitution.

The simplicity means that it is possible to use Robinson's [Rob65] algorithm for first-order, syntactic unification.

In Core-ML, the class of unification problems is more difficult. The difficulty arises from the additional structure in the notion of simple type, and from the context dependent role and scope of type variables:

- The terms to be unified are simple types constructed from an extended grammar including applications of type names $\nu \in \mathit{TypNam}$, and, by implication, definable types $d \in \mathit{DefTyp}$, and types $\tau \in \mathit{Typ}$. Since all of these may themselves contain free simple type variables, the unification algorithm must be extended to these semantic objects.
- The equivalence between simple types is tempered by the equivalence on type names. Unification must take into account the notions of α,η-equivalence of type names, and, by implication, of definable types and types.
- Certain simple type variables $'a \in \mathit{SimTyp}$ and all type variables $\alpha \in \mathit{TypVar}$ must be treated as parameters and cannot be affected, or their scopes violated, by unification.

$\mathcal{Q} \in \mathit{Prefix}$	$::=$	ϵ	empty prefix
	$\mid$	$\mathcal{Q}\exists' a$ (provided $'a \notin \mathcal{V}(\mathcal{Q})$)	simple type meta-variable
	$\mid$	$\mathcal{Q}\forall' a$ (provided $'a \notin \mathcal{V}(\mathcal{Q})$)	simple type parameter
	$\mid$	$\mathcal{Q}\forall\alpha$ (provided $\alpha \notin \mathcal{V}(\mathcal{Q})$)	type parameter

Figure 8.9: Prefixes

Although our class of unification problems is more complex than for algorithm $\mathcal{W}$, it remains within the family of *first-order* unification problems: even though simple types may contain terms with higher-order structure, the meta-variables that are actually game for substitution are restricted to first-order variables ranging over simple types.

To respect the scope of parameters, we will use a first-order variant of Miller's more general algorithm for performing higher-order unification under a mixed prefix of quantifiers [Mil92]. Our unification algorithm will take one additional argument, called the *prefix* to the unification problem. The prefix sets the scene for the unification algorithm by declaring the role and scope of any variables occurring within the terms to be unified.

As in [Mil92], a prefix is simply a sequence of distinct universally and existentially quantified variables:

Definition 8.7 (Prefixes). Figure 8.9 defines the set of prefixes $\mathcal{Q} \in \mathit{Prefix}$. All variables declared in a prefix are required to be distinct.

We let $\mathcal{V}(\mathcal{Q}) \subseteq \mathit{SimTypVar} \cup \mathit{TypVar}$ denote the set of *all* variables declared in the prefix $\mathcal{Q}$; $\mathcal{E}(\mathcal{Q}) \subseteq \mathit{SimTypVar}$ denote the set of variables declared *existentially* in $\mathcal{Q}$; $\mathcal{U}(\mathcal{Q}) \subseteq \mathit{SimTypVar} \cup \mathit{TypVar}$ denote the set of variables declared *universally* in $\mathcal{Q}$. Intuitively, $\mathcal{E}(\mathcal{Q})$ describes the set of *meta-variables* in $\mathcal{Q}$; $\mathcal{U}(\mathcal{Q})$ describes the set of *parameters* in $\mathcal{Q}$.

We will often write a prefix in the form of a *pattern* $\mathcal{Q}\exists' a\mathcal{Q}'$ denoting the concatenation of two prefixes $\mathcal{Q}\exists' a$ and $\mathcal{Q}'$, for some unique $\mathcal{Q}$ and $\mathcal{Q}'$.[2] We will make similar use of the patterns $\mathcal{Q}\forall' a\mathcal{Q}'$ and $\mathcal{Q}\forall\alpha\mathcal{Q}'$.

In the prefix $\mathcal{Q}\exists' a\mathcal{Q}'$ ($\mathcal{Q}\forall' a\mathcal{Q}'$, $\mathcal{Q}\forall\alpha\mathcal{Q}'$), $\mathcal{Q}'$ is the *scope* of the declared variable.

[2]This notation is unambiguous since a variable may be declared at most once in a prefix.

The notation $\mathcal{Q}\bar{\exists}\{'a_0, \ldots, 'a_{n-1}\}$ abbreviates $(\mathcal{Q}\exists' a_0 \cdots)\exists' a_{n-1}$ (in some fixed enumeration of type variables). The abbreviations $\mathcal{Q}\bar{\forall}\{'a_0, \ldots, 'a_{n-1}\}$ and
$\mathcal{Q}\bar{\forall}\{\alpha_0, \ldots, \alpha_{n-1}\}$ are analogous.

A semantic object o is $\mathcal{Q}$-*closed*, written formally as $\mathcal{Q} \vdash o$ **closed**, if, and only if, all of its free variables are declared in $\mathcal{Q}$.

The informal meaning of a prefix as a declaration of variables is as follows. A universal quantifier declares that its bound variable is to be treated as a "constant" that is not available for substitution. Because constants do not usually have scope, we prefer to call universally bound variables *parameters*. An existential quantifier declares that its bound variable is a *meta-variable* and, unlike a parameter, available for substitution. Intuitively, if $'a$ is a meta-variable then any simple type u substituted for $'a$ may contain any of the parameters in scope at the declaration of $'a$ but none of the parameters declared within the scope of $'a$. Concretely, if $'a$ is declared in the prefix $\mathcal{Q}\exists' a\mathcal{Q}'$, then any simple type u substituted for $'a$ may contain any of the parameters declared to the left of $'a$ in $\mathcal{Q}$ but none of the parameters declared to the right of $'a$ in $\mathcal{Q}'$. There is no restriction on the meta-variables that u may contain.

Example 8.3.1. For instance, the prefix $\forall\alpha\exists' a\forall\beta\exists' b$ declares $'a$ and $'b$ as meta-variables and α and β as parameters. A valid substitution term for $'a$ may contain a free occurrence of α, but not of β; a valid substitution term for $'b$ may contain either. Notice that this prefix encodes the side-conditions on parameters arising from the examples in Figures 8.5 through 8.8.

More formally, a prefix determines a set of allowable, or valid, substitutions:

Definition 8.8 ($\mathcal{Q}$-Substitutions). For a given prefix $\mathcal{Q}$ and a substitution σ, we shall say that σ is a $\mathcal{Q}$-*substitution* if, and only if, the relation $\mathcal{Q} \vdash \sigma$ **valid**, defined in Figure 8.10, holds.

It is possible to relate $\mathcal{Q}$-substitutions, according to their generality:

Definition 8.9 ($\mathcal{Q}$-Generality). For a fixed prefix $\mathcal{Q}$, a $\mathcal{Q}$-substitution σ is *more general* than another $\mathcal{Q}$-substitution σ_1, written $\sigma \succeq_{\mathcal{Q}} \sigma_1$, if, and only if,

$$\exists\sigma_2.\sigma_2 \circ \sigma = \sigma_1.$$

We can now define our refined notion of unification problems and their solutions:

$$\boxed{\mathcal{Q} \vdash \sigma\ \textbf{valid}}$$

$$\frac{}{\epsilon \vdash \sigma\ \textbf{valid}} \qquad (\mathcal{Q}\text{-}1)$$

$$\frac{\mathcal{Q} \vdash \sigma\ \textbf{valid}}{\mathcal{Q}\exists' a \vdash \sigma\ \textbf{valid}} \qquad (\mathcal{Q}\text{-}2)$$

$$\frac{\mathcal{Q} \vdash \sigma\ \textbf{valid} \quad 'a \notin \mathrm{Dom}(\sigma) \quad \forall' b \in \mathcal{E}(\mathcal{Q}).'a \notin \mathrm{FTVS}(\sigma('b))}{\mathcal{Q}\forall' a \vdash \sigma\ \textbf{valid}} \qquad (\mathcal{Q}\text{-}3)$$

$$\frac{\mathcal{Q} \vdash \sigma\ \textbf{valid} \quad \forall' b \in \mathcal{E}(\mathcal{Q}).\alpha \notin \mathrm{FV}(\sigma('b))}{\mathcal{Q}\forall \alpha \vdash \sigma\ \textbf{valid}} \qquad (\mathcal{Q}\text{-}4)$$

Figure 8.10: The definition of $\mathcal{Q} \vdash \sigma$ **valid**.

Definition 8.10 ($\mathcal{Q}$-Unification Problems and $\mathcal{Q}$-Unifiers).

A *$\mathcal{Q}$-unification problem* is a triple consisting of a prefix $\mathcal{Q}$ and two $\mathcal{Q}$-closed semantic objects, o and o', drawn from the *same* set of semantic objects. In particular, we assume that either $o, o' \in \mathit{SimTyp}$, $o, o' \in \mathit{DefTyp}^{\mathrm{k}}$, $o, o' \in \mathit{TypNam}^{\kappa}$, or $o, o' \in \mathit{Typ}^{\kappa}$, i.e. that o and o' must have the same kind.

- A *$\mathcal{Q}$-unifier* of o and o' is a $\mathcal{Q}$-substitution σ such that $\sigma(o) = \sigma(o')$. A $\mathcal{Q}$-unifier of o and o' is a *solution* to the $\mathcal{Q}$-unification problem posed by o and o'.
- A *most general $\mathcal{Q}$-unifier* of o and o' is a $\mathcal{Q}$-unifier σ of o and o' such that, for every other $\mathcal{Q}$-unifier σ_1 of o and o', $\sigma \succeq_{\mathcal{Q}} \sigma_1$. A most general $\mathcal{Q}$-unifier of o and o' is a *principal solution* to the $\mathcal{Q}$-unification problem posed by o and o'.

A $\mathcal{Q}$-Unification Algorithm

In this section we will present a deterministic algorithm for constructing a most general $\mathcal{Q}$-unifier for a given $\mathcal{Q}$-unification problem. The input to the algorithm is a valid $\mathcal{Q}$-unification problem, presented as a triple consisting of a prefix $\mathcal{Q}$ and two appropriate $\mathcal{Q}$-closed semantic objects o and

o'. The output, if any, is a most general $\mathcal{Q}$-unifier for the problem. This algorithm is designed to satisfy the following property (cf. Theorem 8.4), whose verification is left to future work:

Property 8.11 (Correctness of $\mathcal{Q}$-Unification). *Provided $\mathcal{Q}$, o and o' define a valid $\mathcal{Q}$-unification problem then:*

Termination *The appeal $\mathcal{Q} \vdash o = o' \downarrow \_$ terminates either in success, returning a substitution, or failure.*

Soundness *If $\mathcal{Q} \vdash o = o' \downarrow \sigma$ then $\mathcal{Q} \vdash \sigma$ **valid** and $\sigma(o) = \sigma(o')$.*

Completeness *If $\sigma_1(o) = \sigma_1(o')$ with $\mathcal{Q} \vdash \sigma_1$ **valid**, then, for some σ, $\mathcal{Q} \vdash o = o' \downarrow \sigma$ with $\sigma \succeq_{\mathcal{Q}} \sigma_1$.*

To express the algorithm, we need to define an additional operation that *updates* a prefix $\mathcal{Q}$ to take into account the effect of applying an intermediate $\mathcal{Q}$-substitution:

Definition 8.12 (Substitution in a Prefix).

$$\begin{array}{rcl}
\_(\_) & \in & (\mathit{Prefix} \times \mathit{Subst}) \rightarrow \mathit{Prefix} \\
\sigma(\epsilon) & \stackrel{\text{def}}{=} & \epsilon \\
\sigma(\mathcal{Q}\exists' a) & \stackrel{\text{def}}{=} & (\sigma(\mathcal{Q}))\bar{\exists}(\text{FTVS}(\sigma('a)) \setminus \mathcal{V}(\sigma(\mathcal{Q}))) \\
\sigma(\mathcal{Q}\forall' a) & \stackrel{\text{def}}{=} & (\sigma(\mathcal{Q}))\forall' a \\
\sigma(\mathcal{Q}\forall\alpha) & \stackrel{\text{def}}{=} & (\sigma(\mathcal{Q}))\forall\alpha
\end{array}$$

Intuitively, if σ is a $\mathcal{Q}$-substitution then $\sigma(\mathcal{Q})$ is a prefix that declares the same set of parameters, and in the same relative order, as $\mathcal{Q}$. Moreover, any variable $'a$ that is not a parameter of $\mathcal{Q}$, but occurs in the image under σ of a meta-variable of $\mathcal{Q}$, is declared as a meta-variable in $\sigma(\mathcal{Q})$ with the following properties: *(a)* for every meta-variable $'b$ of $\mathcal{Q}$ such that $'a \in \text{FTVS}(\sigma('b))$, the scope of $'a$ in $\sigma(\mathcal{Q})$ contains all of the parameters originally declared within the scope of $'b$ in $\mathcal{Q}$; *(b)* $'a$ is declared within the scope of as many parameters as possible without violating property *(a)*. Syntactically speaking, this means that *(a)* for every meta-variable $'b$ of $\mathcal{Q}$ such that $'a \in \text{FTVS}(\sigma('b))$, $'a$ is existentially quantified to the left of any parameters originally declared to the right of $'b$; *(b)* $'a$ is existentially quantified as far to the right as possible without violating *(a)*.

The formal motivation for defining substitution in a prefix is the following lemma:

Lemma 8.13 (Composition). *Let σ_1 be a $\mathcal{Q}$-substitution. Then, for any other substitution σ_2, $\sigma_2 \circ \sigma_1$ is a $\mathcal{Q}$-substitution if, and only if, σ_2 is a $\sigma_1(\mathcal{Q})$-substitution.*

The forward direction of this lemma can be used to prove the soundness of our unification algorithm: namely, that the composite substitutions returned by the algorithm are valid for the original prefix of the unification problem. The reverse direction can be used to prove completeness: updating the prefix in recursive calls does not exclude valid unifiers.

The following properties allow us to show that the unification algorithm only invokes itself on valid unification problems in recursive calls; and that it returns a valid substitution for the original prefix in rules that extend the current prefix with new parameters.

Properties 8.14 (Closure and Weakening).

1. *If* $\mathcal{Q} \vdash o$ **closed**, $\mathcal{Q} \vdash \sigma$ **valid**, *and* $\forall' a \in \mathcal{E}(\mathcal{Q}).\mathrm{FV}(\sigma('a)) \subseteq \mathcal{U}(\mathcal{Q})$ *then* $\sigma(\mathcal{Q}) \vdash \sigma(o)$ **closed**.
2. *If* $\mathcal{Q}\forall' a \vdash \sigma$ **valid** *then* $\mathcal{Q} \vdash \sigma$ **valid**.
3. *If* $\mathcal{Q}\forall\alpha \vdash \sigma$ **valid** *then* $\mathcal{Q} \vdash \sigma$ **valid**.

We can now present our unification algorithm:

Unification of Simple Types $\boxed{\mathcal{Q} \vdash u = u' \downarrow \sigma}$

$$\frac{}{\mathcal{Q}\forall' a\mathcal{Q}' \vdash {'a} = {'a} \downarrow \emptyset} \qquad (\mathcal{U}\text{-}1)$$

$$\frac{}{\mathcal{Q}\exists' a\mathcal{Q}' \vdash {'a} = {'a} \downarrow \emptyset} \qquad (\mathcal{U}\text{-}2)$$

($\mathcal{U}$-1),($\mathcal{U}$-2) A simple type variable, whether a parameter or a meta-variable, unifies with itself under the empty substitution.

$$\frac{\mathcal{Q}\bar{\exists}\mathcal{E}(\mathcal{Q}') \vdash u \ \textbf{closed}}{\mathcal{Q}\exists' a\mathcal{Q}' \vdash {'a} = u \downarrow [u/'a]} \qquad (\mathcal{U}\text{-}3)$$

$$\frac{\mathcal{Q}\bar{\exists}\mathcal{E}(\mathcal{Q}') \vdash u \ \textbf{closed} \quad \forall' b \in \mathcal{E}(\mathcal{Q}\mathcal{Q}').u \not\equiv {'b}}{\mathcal{Q}\exists' a\mathcal{Q}' \vdash u = {'a} \downarrow [u/'a]} \qquad (\mathcal{U}\text{-}4)$$

($\mathcal{U}$-3),($\mathcal{U}$-4) The rules are almost symmetric. Assuming that u is $\mathcal{Q}\exists' a\mathcal{Q}'$-closed, the premise $\mathcal{Q}\bar{\exists}\mathcal{E}(\mathcal{Q}') \vdash u$ **closed** merely requires that none of the parameters declared within the scope of the meta-variable $'a$ occur free in u, and that $'a$ does not occur free in u. The former ensures that $[u/'a]$ is a $\mathcal{Q}\exists' a\mathcal{Q}'$-substitution as well as a unifier. The latter is just the *occur check* that we already encountered in Rules ($\mathcal{R}$-2) and ($\mathcal{R}$-3) of Robinson's algorithm. The side condition $\forall' b \in \mathcal{E}(\mathcal{Q}\mathcal{Q}').u \not\equiv {}'b$ on Rule ($\mathcal{U}$-4) ensures that only Rule ($\mathcal{U}$-3) applies in the special case that both types are distinct meta-variables. This side condition plays the same role as the second side condition of Rule ($\mathcal{R}$-3).

$$\frac{\mathcal{Q} \vdash u_1 = u_1' \downarrow \sigma_1 \quad \sigma_1(\mathcal{Q}) \vdash \sigma_1(u_2) = \sigma_1(u_2') \downarrow \sigma_2}{\mathcal{Q} \vdash u_1 \to u_2 = u_1' \to u_2' \downarrow \sigma_2 \circ \sigma_1} \quad (\mathcal{U}\text{-}5)$$

($\mathcal{U}$-5) To unify two function spaces, we first attempt to unify their domains to obtain a $\mathcal{Q}$-substitution σ_1. Provided this succeeds, we then attempt to unify $\sigma_1(u_2)$ and $\sigma_1(u_2')$ with respect to the updated prefix $\sigma_1(\mathcal{Q})$. Note that revising the prefix ensures that σ_2 is a $\sigma_1(\mathcal{Q})$-substitution and thus that $\sigma_2 \circ \sigma_1$ is a $\mathcal{Q}$-substitution (cf. Lemma 8.13 (Composition)).

$$\frac{\begin{array}{l}\mathcal{Q} \vdash \nu = \nu' \downarrow \sigma \\ \forall i \in [\mathrm{k}].\ \big(\bar{\sigma}(\mathcal{Q}) \vdash \bar{\sigma}(u_i) = \bar{\sigma}(u_i') \downarrow \sigma_i \text{ where } \bar{\sigma} \equiv (\sigma_{(i-1)} \circ \ldots \sigma_0 \circ \sigma)\big)\end{array}}{\mathcal{Q} \vdash \nu(u_0, \ldots, u_{(\mathrm{k}-1)}) = \nu'(u_0', \ldots, u_{(\mathrm{k}-1)}') \downarrow \sigma_{(\mathrm{k}-1)} \circ \ldots \sigma_0 \circ \sigma} \quad (\mathcal{U}\text{-}6)$$

($\mathcal{U}$-6) To unify two type name applications, we first attempt to unify their type names, and then iteratively attempt to unify their corresponding arguments. Each subsequent appeal to the algorithm must take into account the substitutions returned by previous appeals. The rule only applies when the two type names both have the same number of arguments and are thus of the same kind (unification must fail otherwise).

Unification of Definable Types $\boxed{\mathcal{Q} \vdash d = d' \downarrow \sigma}$

$$\frac{\begin{array}{l}\forall i \in [\mathrm{k}].'a_i \not\in \mathcal{V}(\mathcal{Q}) \cup \{'a_0, \ldots, 'a_{(i-1)}\} \\ \mathcal{Q}\bar{\forall}'a_0, \ldots, 'a_{(\mathrm{k}-1)} \vdash u = \{'b_i \mapsto 'a_i | i \in [\mathrm{k}]\}\,(u') \downarrow \sigma\end{array}}{\mathcal{Q} \vdash \Lambda('a_0, \ldots, 'a_{(\mathrm{k}-1)}).u = \Lambda('b_0, \ldots, 'b_{(\mathrm{k}-1)}).u' \downarrow \sigma} \quad (\mathcal{U}\text{-}7)$$

($\mathcal{U}$-7) To unify two definable types, we first rename the formal arguments of one to coincide with the formal arguments of the other and declare them as fresh parameters in the prefix. The premise

$$\forall i \in [\mathrm{k}].{}'a_i \notin \mathcal{V}(\mathcal{Q}) \cup \{{}'a_0, \ldots, {}'a_{(i-1)}\}$$

together with the assumption that $\Lambda({}'b_0, \ldots, {}'b_{(\mathrm{k}-1)}).u'$ is $\mathcal{Q}$-closed ensures that this renaming does not capture any free variables of $\Lambda({}'b_0, \ldots, {}'b_{(\mathrm{k}-1)}).u'$. We then unify the bodies in the extended prefix. Observe that, because the parameters are declared to the right of any meta-variable in $\mathcal{Q}$, the parameters are prevented from occurring in the image of any meta-variable under σ. Moreover, being parameters, they cannot occur in the domain of σ. It is easy to reason that, under these conditions, if σ is a unifier of u and $\{{}'b_i \mapsto {}'a_i | i \in [\mathrm{k}]\}\,(u')$ then σ is also a unifier of $\Lambda({}'a_0, \ldots, {}'a_{(\mathrm{k}-1)}).u$ and $\Lambda({}'b_0, \ldots, {}'b_{(\mathrm{k}-1)}).u'$. Note also that the rule applies only when the two definable types take the same number of arguments and are thus of the same kind (unification must fail otherwise).

Unification of Type Names $\boxed{\mathcal{Q} \vdash \nu = \nu' \downarrow \sigma}$

$$\frac{}{\mathcal{Q}\forall\alpha\,\mathcal{Q}' \vdash \alpha = \alpha \downarrow \emptyset} \qquad (\mathcal{U}\text{-8})$$

($\mathcal{U}$-8) A type variable must be declared as a parameter. It unifies with itself under the empty substitution.

$$\frac{\begin{array}{l}\nu, \nu' \in \mathit{TypNam}^{\kappa} \\ \mathcal{Q} \vdash \nu = \nu' \downarrow \sigma_1 \\ \sigma_1(\mathcal{Q}) \vdash \sigma_1(\tau) = \sigma_1(\tau') \downarrow \sigma_2\end{array}}{\mathcal{Q} \vdash \nu\ \tau = \nu'\ \tau' \downarrow \sigma_2 \circ \sigma_1} \qquad (\mathcal{U}\text{-9})$$

($\mathcal{U}$-9) To unify two type applications we proceed essentially as in rule ($\mathcal{U}$-6). However, we need to first ensure that the type names in the left and right hand sides are of the same kind (unification must fail if they are not).

Unification of Types $\boxed{\mathcal{Q} \vdash \tau = \tau' \downarrow \sigma}$

$$\frac{\alpha \notin \mathcal{V}(\mathcal{Q}) \quad \mathcal{Q}\forall\alpha \vdash \tau = [\alpha/\beta]\,(\tau') \downarrow \sigma}{\mathcal{Q} \vdash \Lambda\alpha.\tau = \Lambda\beta.\tau' \downarrow \sigma} \qquad (\mathcal{U}\text{-10})$$

($\mathcal{U}$-10) In a manner similar to Rule $\mathcal{U}$-7, to unify two type abstractions, we first rename the formal argument of one to coincide with the argument of the other and declare it as a fresh parameter in the prefix. The premise $\alpha \notin \mathcal{V}(\mathcal{Q})$ together with the assumption that $\Lambda\beta.\tau'$ is $\mathcal{Q}$-closed ensures that this renaming does not capture any free variables of $\Lambda\beta.\tau'$. We then unify the bodies in the extended prefix. Declaring the parameter to the right of any meta-variables in $\mathcal{Q}$ prevents it from occurring in their image under σ. Under these conditions, it is easy to reason that if σ is a unifier of τ and $[\beta/\alpha]\,(\tau')$ then σ is also a unifier of $\Lambda\alpha.\tau$ and $\Lambda\beta.\tau'$.

$$\frac{\mathcal{Q} \vdash \nu = \nu' \downarrow \sigma}{\mathcal{Q} \vdash \nu = \nu' \downarrow \sigma} \qquad (\mathcal{U}\text{-}11)$$

($\mathcal{U}$-11) If both types are type names, we simply unify the type names.

$$\frac{\alpha \notin \mathcal{V}(\mathcal{Q}) \quad \mathcal{Q}\forall\alpha \vdash \tau = \nu\ \alpha \downarrow \sigma}{\mathcal{Q} \vdash \Lambda\alpha.\tau = \nu \downarrow \sigma} \qquad (\mathcal{U}\text{-}12)$$

$$\frac{\alpha \notin \mathcal{V}(\mathcal{Q}) \quad \mathcal{Q}\forall\alpha \vdash \nu\ \alpha = \tau \downarrow \sigma}{\mathcal{Q} \vdash \nu = \Lambda\alpha.\tau \downarrow \sigma} \qquad (\mathcal{U}\text{-}13)$$

($\mathcal{U}$-12),($\mathcal{U}$-13) These rules are symmetric. If one of the types is an abstraction, but the other is a type name, then unification does not immediately fail, for there may be a unifier that makes the two terms η-equivalent. (Recall that Definition 5.2 requires that we identify any type of the form $\Lambda\alpha.\nu\ \alpha \in \mathit{TypNam}^{\kappa\to\kappa'}$ with its η-contraction ν, provided $\alpha \notin \mathrm{FV}(\nu)$ and $\nu \in \mathit{TypNam}^{\kappa\to\kappa'}$.) Note that, provided ν is $\mathcal{Q}$-closed, the first premise ensures that α does not occur in $\mathrm{FV}(\nu)$. Hence $\Lambda\alpha.\nu\ \alpha$ is an η-expansion of ν. Our algorithm proceeds in the recursive call by declaring the shared parameter and unifying the body of the original abstraction with the body of the η-expansion. This trick for dealing with η-equivalence is inspired by Coquand's algorithm for testing conversion in Type Theory [Coq91].

$$\frac{\mathcal{Q} \vdash d = d' \downarrow \sigma}{\mathcal{Q} \vdash d = d' \downarrow \sigma} \qquad (\mathcal{U}\text{-}14)$$

($\mathcal{U}$-14) If both types are definable types, we simply unify the definable types.

$$\frac{\mathcal{Q} \vdash d = \eta(\nu) \ \downarrow \ \sigma}{\mathcal{Q} \vdash d = \nu \ \downarrow \ \sigma} \qquad (\mathcal{U}\text{-}15)$$

$$\frac{\mathcal{Q} \vdash \eta(\nu) = d \ \downarrow \ \sigma}{\mathcal{Q} \vdash \nu = d \ \downarrow \ \sigma} \qquad (\mathcal{U}\text{-}16)$$

($\mathcal{U}$-15),($\mathcal{U}$-16) These rules are symmetric and similar to Rules ($\mathcal{U}$-12) and ($\mathcal{U}$-13). If one of the types is a definable type, but the other is a type name, then unification does not immediately fail, for there may be a unifier that makes the two terms η-equivalent. (Recall that Definition 5.2 requires that we identify any type name with its η-expansion as a definable type.) In the recursive call, we first η-expand the type name to obtain a definable type, and then unify the two definable types.

Example 8.3.2. For example, consider the problem of finding a most general $\forall\alpha\exists' a\forall\beta\exists' b$-unifier of $'a \to 'b$ and $'b \to 'a$. The corresponding appeal to the algorithm:

$$\frac{\dfrac{\forall\alpha\exists' b \vdash {}'b \ \textbf{closed}}{\forall\alpha\exists' a\forall\beta\exists' b \vdash {}'a = {}'b \ \downarrow \ [{}'b/{}'a]}\,(\mathcal{U}\text{-}3) \quad \dfrac{}{\forall\alpha\exists' b\forall\beta \vdash {}'b = {}'b \ \downarrow \ \emptyset}\,(\mathcal{U}\text{-}2)}{\forall\alpha\exists' a\forall\beta\exists' b \vdash {}'a \to {}'b = {}'b \to {}'a \ \downarrow \ [{}'b/{}'a]}\,(\mathcal{U}\text{-}5)$$

returns the substitution $[{}'b/{}'a]$. It is easy to see that $[{}'b/{}'a]$ is a $\forall\alpha\exists' a\forall\beta\exists' b$-unifier of $'a \to {}'b$ and $'b \to {}'a$, since

$$[{}'b/{}'a]({}'a \to {}'b) = {}'b \to {}'b = [{}'b/{}'a]({}'b \to {}'a)$$

and $\beta \notin \mathrm{FV}([{}'b/{}'a]({}'a))$. To see that it is a most general unifier, consider any other $\forall\alpha\exists' a\forall\beta\exists' b$-unifier σ_1 of $'a \to {}'b$ and $'b \to {}'a$. Then we must have $\sigma_1({}'a) = \sigma_1({}'b)$ with $\beta \notin \mathrm{FV}(\sigma_1({}'a))$. Let $\sigma_2 = \sigma_1$, then it is easy to see that $\sigma_2 \circ [{}'b/{}'a] = \sigma_1$, i.e. $[{}'b/{}'a] \succeq_{\forall\alpha\exists' a\forall\beta\exists' b} \sigma_1$.

Example 8.3.3. For another example, consider the problem of finding a most general $\forall\alpha\exists' a\forall\beta\exists' b$unifier of $'a \to {}'b$ and $'b \to \beta$. The corresponding appeal to the algorithm:

$$\frac{\dfrac{\forall\alpha\exists' b \vdash {}'b \ \textbf{closed}}{\forall\alpha\exists' a\forall\beta\exists' b \vdash {}'a = {}'b \ \downarrow \ [{}'b/{}'a]}\,(\mathcal{U}\text{-}3) \quad \dfrac{\forall\alpha \vdash \beta \ \textbf{closed fails}}{\forall\alpha\exists' b\forall\beta \vdash {}'b = \beta \ \downarrow}\,(\mathcal{U}\text{-}3)}{\forall\alpha\exists' a\forall\beta\exists' b \vdash {}'a \to {}'b = {}'b \to \beta \ \downarrow}\,(\mathcal{U}\text{-}5)$$

fails to return a substitution, because the condition $\forall\alpha \vdash \beta$ **closed** does not hold at the root of the rightmost recursive call. But this is fine, since there is no $\forall\alpha\exists' a\forall\beta\exists' b$-unifier of $'a \to {}'b$ and $'b \to \beta$. Suppose, to the contrary, that there was such a unifier σ. Then we must have $\sigma('a \to 'b) = \sigma('b \to \beta)$ with $\beta \notin \mathrm{FV}(\sigma('a))$. This just means that $\sigma('a) = \sigma('b)$ and $\sigma('b) = \beta$. But then $\mathrm{FV}(\sigma('a)) = \{\beta\}$ which contradicts $\beta \notin \mathrm{FV}(\sigma('a))$.

8.3.2 Value Type Enrichment modulo Unification

In Rule (M-4) of the signature matching algorithm (Definition 5.25), we implicitly assumed the existence of a decidable test for determining whether one value type enriches another, i.e. for checking whether the relation $v \succeq v'$ holds between two value types of the generic Core language. For Core-ML, Definition 3.28 defines $v \succeq v'$ to hold if, and only if, "every simple type generalised by v' is also generalised by v". This definition is intuitive, but the infinitary condition prevents it from directly yielding a decision procedure.

Fortunately, it can be shown that v enriches v' if, and only if, v generalises a *generic instance* of v' [Tof88]. Formally, let $v \equiv \forall 'a_0, \ldots, 'a_{(m-1)}.u$ and $v' \equiv \forall 'b_0, \ldots, 'b_{(n-1)}.u'$. Provided that $\{'b_0, \ldots, 'b_{(n-1)}\} \cap \mathrm{FTVS}(v) = \emptyset$ then $v \succeq v'$ if, and only if, $v \succ u'$, i.e. there exists a substitution σ with $\mathrm{Dom}(\sigma) = \{'a_0, \ldots, 'a_{(m-1)}\}$ such that $\sigma(u) = u'$.

Moreover, since we can w.l.o.g. also assume that $\{'a_0, \ldots, 'a_{(m-1)}\} \cap \mathrm{FTVS}(u') = \emptyset$, we can easily decide whether $v \succ u'$ by looking for a unifier σ with $\mathrm{Dom}(\sigma) = \{'a_0, \ldots, 'a_{(m-1)}\}$ such that $\sigma(u) = \sigma(u') = u'$. Thus the test for $v \succeq v'$ can be posed as the following $\mathcal{Q}$-unification problem. Set $\mathcal{Q}$ to be the prefix that declares all the free variables of v and v' as parameters. Then $v \succeq v'$ if, and only if, u and u' have a $\mathcal{Q}\bar{\forall}\{'b_0, \ldots, 'b_{(n-1)}\}\bar{\exists}\{'a_0, \ldots, 'a_{(m-1)}\}$-unifier. Clearly, this can be decided by a corresponding appeal to our unification algorithm.

It is not difficult to argue that, in the more general case, where $\mathcal{Q}$ declares some of the simple type variables free in v and v', not as parameters, but as *meta-variables*, then the successful appeal

$$\mathcal{Q}\bar{\forall}\{'b_0, \ldots, 'b_{(n-1)}\}\bar{\exists}\{'a_0, \ldots, 'a_{(m-1)}\} \vdash u = u' \downarrow \sigma$$

can be used to obtain a most general $\mathcal{Q}$-substitution σ' such that $\sigma'(v) \succeq \sigma'(v')$, by choosing $\sigma' = \sigma \setminus \{'a_0, \ldots, 'a_{(m-1)}\}$.

This idea is embodied in the following algorithm which we shall need shortly.

Value Type Matching $\boxed{\mathcal{Q} \vdash v \succeq v' \downarrow \sigma}$

$$\frac{\begin{array}{l}\forall i \in [n].{'b_i} \notin \mathcal{V}(\mathcal{Q}) \cup \{'b_0, \ldots, 'b_{(i-1)}\} \\ \forall i \in [m].{'a_i} \notin \mathcal{V}(\mathcal{Q}) \cup \{'b_0, \ldots, 'b_{(n-1)}\} \cup \{'a_0, \ldots, 'a_{(i-1)}\} \\ \mathcal{Q}\bar{\forall}\{'b_0, \ldots, 'b_{(n-1)}\}\bar{\exists}\{'a_0, \ldots, 'a_{(m-1)}\} \vdash u = u' \downarrow \sigma\end{array}}{\mathcal{Q} \vdash \forall'a_0, \ldots, 'a_{(m-1)}.u \succeq \forall'b_0, \ldots, 'b_{(n-1)}.u' \downarrow \sigma \setminus \{'a_0, \ldots, 'a_{(m-1)}\}} \quad (\mathcal{V}\text{-}1)$$

It is designed to satisfy the following property, whose verification is left to future work:

Property 8.15 (Correctness of Value Type Matching). *For any prefix $\mathcal{Q}$ and $\mathcal{Q}$-closed value types v and v':*

Termination *The appeal $\mathcal{Q} \vdash v \succeq v' \downarrow \_$ terminates either in success, returning a substitution, or failure.*

Soundness *If $\mathcal{Q} \vdash v \succeq v' \downarrow \sigma$ then $\mathcal{Q} \vdash \sigma$* **valid** *and $\sigma(v) \succeq \sigma(v')$.*

Completeness *If $\sigma_1(v) \succeq \sigma_1(v')$ with $\mathcal{Q} \vdash \sigma_1$* **valid**, *then, for some σ, $\mathcal{Q} \vdash v \succeq v' \downarrow \sigma$ with $\sigma \succeq_{\mathcal{Q}} \sigma_1$.*

Example 8.3.4. Unlike the rules of the unification algorithm, that can at most declare additional *parameters* in recursive calls, the algorithm for value type matching can declare additional *meta-variables* before invoking unification. For instance, consider the derivation:

$$\cfrac{\cfrac{\begin{array}{c}\mathcal{Q}\bar{\exists}\mathcal{E}(\mathcal{Q}'\exists'c_1\exists'c_2) \vdash 'c_1 \to 'c_2 \ \mathbf{closed} \\ \forall'd \in \mathcal{E}(\mathcal{Q}\mathcal{Q}'\exists'c_1\exists'c_2).'c_1 \to 'c_2 \not\equiv 'd\end{array}}{\mathcal{Q}\exists'a\mathcal{Q}'\exists'c_1\exists'c_2 \vdash 'c_1 \to 'c_2 = 'a \downarrow ['c_1 \to 'c_2/'a]} \ (\mathcal{U}\text{-}4)}{\mathcal{Q}\exists'a\mathcal{Q}' \vdash \forall'b_1, 'b_2.'b_1 \to 'b_1 \succeq \forall\emptyset.'a \downarrow ['c_1 \to 'c_2/'a]} \ (\mathcal{V}\text{-}1)$$

The recursive call to unification extends the prefix with two new meta-variables $'c_1$ and $'c_2$. This derivation is sound because:

$$\begin{array}{rcl}['c_1 \to 'c_2/'a](\forall'b_1, 'b_2.'b_1 \to 'b_2) & \equiv & \forall'b_1, 'b_2.'b_1 \to 'b_2 \\ & \succeq & \forall\emptyset.'c_1 \to 'c_2 \\ & \equiv & ['c_1 \to 'c_2/'a](\forall\emptyset.'a)\end{array}$$

Notice that updating the original prefix $\mathcal{Q}\exists'a\mathcal{Q}'$ by the resulting substitution $['c_1 \to 'c_2/'a]$ returns the modified prefix $\mathcal{Q}\exists'c_1\exists'c_2\mathcal{Q}'$ that declares not one but *two* fresh meta-variables $'c_1$ and $'c_2$, whose declarations replace the single declaration of $'a$.

8.3.3 Matching modulo Unification

In Chapter 5 we presented an algorithm for "matching" one module type against another. Provided suitable conditions hold of the inputs P, R, $\mathcal{O}$ and $\mathcal{O}'$, the appeal $\forall P.\forall R \vdash \mathcal{O} \succeq \mathcal{O}' \ \downarrow \ \_$ produces as output a unique matching realisation φ if, and only if, $\mathcal{O} \succeq \varphi\left(\mathcal{O}'\right)$. In the context of Core-ML type inference, we can encounter matching problems in which both $\mathcal{O}$ and $\mathcal{O}'$ contain meta-variables, introduced by the type inference algorithm. In this situation, the matching problems we need to solve are more general. In particular, the typical matching problem will be to find both a substitution σ and a realisation φ such $\sigma(\mathcal{O}) \succeq \varphi\left(\sigma(\mathcal{O}')\right)$. Clearly, the algorithm $\forall P.\forall R \vdash \mathcal{O} \succeq \mathcal{O}' \ \downarrow \ \_$ as it stands is not appropriate. Fortunately, we can adapt it to the more general setting.

Our generalised matching algorithm, that combines the search for a substitution with the construction of a matching realisation, is derived from the matching algorithm in Chapter 5. Successful appeals to the algorithm have the form $\mathcal{Q}.\forall R \vdash \mathcal{O} \succeq \mathcal{O}' \ \downarrow \ \sigma, \varphi$. The four arguments $\mathcal{Q}$, R, $\mathcal{O}$ and $\mathcal{O}'$ play essentially the same role as in the original algorithm, except that the first has been generalised from a set of type variables P to a prefix $\mathcal{Q}$. The algorithm outputs both a substitution σ and, as before, a realisation φ. Intuitively, the matching algorithm constructs a most general $\mathcal{Q}$-substitution σ and realisation φ such that $\sigma(\mathcal{O}) \succeq \varphi\left(\sigma(\mathcal{O}')\right)$ (provided they exist).

In the original algorithm, the first argument P merely records the set of type variables, other than R, allowed to occur free in the objects $\mathcal{O}$ and $\mathcal{O}'$. In the generalised algorithm, $\mathcal{Q}$ subsumes the role of P. Like P, $\mathcal{Q}$ records the set of type variables, other than R, allowed to occur free in both objects $\mathcal{O}$ and $\mathcal{O}'$, by declaring these variables as parameters. However, it also sets the scene for any unification problems encountered during matching by declaring the role and scope of any simple type variable occurring free in $\mathcal{O}$ and $\mathcal{O}'$.

To see how the algorithm is derived from the algorithm in Chapter 5 we shall compare the original rule, Rule (M-2):

$$\frac{\mathrm{t} \in \mathrm{Dom}(\mathcal{S}) \quad \mathcal{S}(\mathrm{t}) = \tau \quad \forall P.\forall R \vdash \mathcal{S} \succeq \mathcal{S}' \downarrow \varphi}{\forall P.\forall R \vdash \mathcal{S} \succeq \mathrm{t} = \tau, \mathcal{S}' \downarrow \varphi}$$

with its generalisation, Rule ($\mathcal{M}$-2):

$$\frac{\begin{array}{l} \mathrm{t} \in \mathrm{Dom}(\mathcal{S}) \\ \mathcal{S}(\mathrm{t}), \tau \in \mathit{Typ}^{\kappa} \\ \mathcal{Q}\bar{\forall} R \vdash \tau \ \mathbf{closed} \\ Q\bar{\forall} R \vdash \mathcal{S}(\mathrm{t}) = \tau \ \downarrow \ \sigma_1 \\ \sigma_1(\mathcal{Q}).\forall R \vdash \sigma_1(\mathcal{S}) \succeq \sigma_1(\mathcal{S}') \ \downarrow \ \sigma_2, \varphi \end{array}}{\mathcal{Q}.\forall R \vdash \mathcal{S} \succeq \mathrm{t} = \tau, \mathcal{S}' \ \downarrow \ \sigma_2 \circ \sigma_1, \varphi}$$

In the generalised rule, the original test for type equivalence, $\mathcal{S}(\mathrm{t}) = \tau$, has been replaced by a test for unifiability, expressed by the three premises:

$$\mathcal{S}(\mathrm{t}), \tau \in \mathit{Typ}^{\kappa}$$

$$\mathcal{Q}\bar{\forall} R \vdash \tau \ \mathbf{closed}$$

$$Q\bar{\forall} R \vdash \mathcal{S}(\mathrm{t}) = \tau \ \downarrow \ \sigma_1$$

Of these, the first premise merely states that the two type components are of the same kind, which is a necessary precondition on the inputs to the unification algorithm[3]. The second premise ensures that the type τ is closed with respect to the variables allowed to occur free in $\mathcal{S}(\mathrm{t})$. This premise discriminates between applications of this rule and Rule ($\mathcal{M}$-3)[4]. The third premise, the appeal to the unification algorithm, is the generalisation proper of the original premise $\mathcal{S}(\mathrm{t}) = \tau$. If the components can be unified, the unifier σ_1 is applied to the structures $\mathcal{S}$ and $\mathcal{S}'$ before proceeding with the recursive call. The result, if any, is a second substitution σ_2 and matching realisation φ such that $\sigma_2(\sigma_1(\mathcal{S})) \succeq \varphi(\sigma_2(\sigma_1(\mathcal{S})))$. The most general substitution for the original problem is obtained by composing σ_2 and σ_1. The matching realisation is just the realisation φ.

The only other rule that can actually contribute to the final substitution returned by matching, rather than merely propagating substitutions obtained from recursive calls, is Rule ($\mathcal{M}$-4), the generalisation of Rule (M-4). The rule determines whether the corresponding value component of the left-hand structure enriches the value component of the right-hand structure, modulo some most general substitution. This is decided by an appeal to the algorithm $\mathcal{Q} \vdash v \succeq v' \ \downarrow \ \_$ of the preceding section.

[3]In the original rule, this premise is implicit because Definition 5.2 stipulates that type equivalence is only defined on types of the same kind.

[4]In the original rule, this premise is implicit because two types can only be equivalent if they contain the same free variables.

The remaining rules are straightforward adaptations of the corresponding originals: each rule has been altered to propagate the additional substitution from the output of one recursive call to the inputs of the next, and to return the appropriate composite substitution and realisation, taking care to apply any intervening substitution to a previously computed component of this realisation. (The result of applying a substitution to a realisation is defined in the obvious way: $\sigma(\varphi) \stackrel{\text{def}}{=} \{\alpha \mapsto \sigma(\varphi(\alpha)) \mid \alpha \in \text{Dom}(\varphi)\}$.)

Structure Matching $\boxed{\mathcal{Q}.\forall R \vdash \mathcal{S} \succeq \mathcal{S}' \downarrow \sigma, \varphi}$

$$\frac{}{\mathcal{Q}.\forall R \vdash \mathcal{S} \succeq \epsilon_{\mathcal{S}} \downarrow \emptyset, \emptyset} \quad (\mathcal{M}\text{-1})$$

$$\frac{\begin{array}{l} \text{t} \in \text{Dom}(\mathcal{S}) \\ \mathcal{S}(\text{t}), \tau \in \mathit{Typ}^{\kappa} \\ \mathcal{Q}\bar{\forall} R \vdash \tau \textbf{ closed} \\ Q\bar{\forall} R \vdash \mathcal{S}(\text{t}) = \tau \downarrow \sigma_1 \\ \sigma_1(\mathcal{Q}).\forall R \vdash \sigma_1(\mathcal{S}) \succeq \sigma_1(\mathcal{S}') \downarrow \sigma_2, \varphi \end{array}}{\mathcal{Q}.\forall R \vdash \mathcal{S} \succeq \text{t} = \tau, \mathcal{S}' \downarrow \sigma_2 \circ \sigma_1, \varphi} \quad (\mathcal{M}\text{-2})$$

$$\frac{\begin{array}{l} \alpha \notin \mathcal{V}(\mathcal{Q}) \cup R \\ \text{t} \in \text{Dom}(\mathcal{S}) \\ \mathcal{S}(\text{t}), (\alpha\ \beta_0 \cdots \beta_{n-1}) \in \mathit{Typ}^{\kappa} \\ \text{FV}(\mathcal{S}(\text{t})) \cap R \subseteq \{\beta_i | i \in [n]\} \\ \mathcal{Q}.\forall R \vdash \mathcal{S} \succeq [\Lambda\beta_0 \cdots \beta_{n-1}.\mathcal{S}(\text{t})/\alpha]\,(\mathcal{S}') \downarrow \sigma, \varphi \end{array}}{\mathcal{Q}.\forall R \vdash \mathcal{S} \succeq \text{t} = \alpha\ \beta_0 \cdots \beta_{n-1}, \mathcal{S}' \downarrow \sigma, (\sigma([\Lambda\beta_0 \cdots \beta_{n-1}.\mathcal{S}(\text{t})/\alpha]) \mid \varphi)} \quad (\mathcal{M}\text{-3})$$

$$\frac{\begin{array}{l} \text{x} \in \text{Dom}(\mathcal{S}) \\ Q\bar{\forall} R \vdash \mathcal{S}(\text{x}) \succeq v \downarrow \sigma_1 \\ \sigma_1(\mathcal{Q}).\forall R \vdash \sigma_1(\mathcal{S}) \succeq \sigma_1(\mathcal{S}') \downarrow \sigma_2, \varphi \end{array}}{\mathcal{Q}.\forall R \vdash \mathcal{S} \succeq \text{x} : v, \mathcal{S}' \downarrow \sigma_2 \circ \sigma_1, \varphi} \quad (\mathcal{M}\text{-4})$$

$$\frac{\begin{array}{l} \text{X} \in \text{Dom}(\mathcal{S}) \\ \mathcal{Q}.\forall R \vdash \mathcal{S}(\text{X}) \succeq \mathcal{M} \downarrow \sigma_1, \varphi \\ \sigma_1(\mathcal{Q}).\forall R \vdash \sigma_1(\mathcal{S}) \succeq \varphi\,(\sigma_1(\mathcal{S}')) \downarrow \sigma_2, \varphi' \end{array}}{\mathcal{Q}.\forall R \vdash \mathcal{S} \succeq \text{X} : \mathcal{M}, \mathcal{S}' \downarrow \sigma_2 \circ \sigma_1, (\sigma_2(\varphi) \mid \varphi')} \quad (\mathcal{M}\text{-5})$$

Functor Matching $\boxed{\mathcal{Q}.\forall R \vdash \mathcal{F} \succeq \mathcal{F}' \downarrow \sigma, \varphi}$

$$\frac{\begin{array}{l} N \cap (\mathcal{V}(\mathcal{Q}) \cup R \cup M) = \emptyset \\ M \cap (\mathcal{V}(\mathcal{Q}) \cup R) = \emptyset \\ (\mathcal{Q}\bar{\forall} R \cup M).\forall\emptyset \vdash \mathcal{M}_M \succeq \mathcal{M}_N \downarrow \sigma_1, \varphi' \\ \sigma_1(\mathcal{Q}).\forall R \cup M \vdash \varphi'\,(\sigma_1(\mathcal{M}'_N)) \succeq \sigma_1(\mathcal{M}'_M) \downarrow \sigma_2, \varphi \end{array}}{\mathcal{Q}.\forall R \vdash \forall N.\mathcal{M}_N \to \mathcal{M}'_N \succeq \forall M.\mathcal{M}_M \to \mathcal{M}'_M \downarrow \sigma_2 \circ \sigma_1, \varphi} \qquad (\mathcal{M}\text{-}6)$$

Module Matching $\boxed{\mathcal{Q}.\forall R \vdash \mathcal{M} \succeq \mathcal{M}' \downarrow \sigma, \varphi}$

$$\frac{\mathcal{Q}.\forall R \vdash \mathcal{S} \succeq \mathcal{S}' \downarrow \sigma, \varphi}{\mathcal{Q}.\forall R \vdash \mathcal{S} \succeq \mathcal{S}' \downarrow \sigma, \varphi} \qquad (\mathcal{M}\text{-}7)$$

$$\frac{\mathcal{Q}.\forall R \vdash \mathcal{F} \succeq \mathcal{F}' \downarrow \sigma, \varphi}{\mathcal{Q}.\forall R \vdash \mathcal{F} \succeq \mathcal{F}' \downarrow \sigma, \varphi} \qquad (\mathcal{M}\text{-}8)$$

The generalised matching algorithm is designed to satisfy the following property (cf. Theorems 5.26 (Termination), 5.32 (Soundness) and 5.33 (Completeness)), whose verification is left to future work:

Property 8.16 (Correctness of Matching). *On inputs $\mathcal{Q}$, R, $\mathcal{O}$ and $\mathcal{O}'$, where $P \equiv \mathcal{U}(\mathcal{Q}) \cap TypVar$ and for any Q such that $\forall P \cup R \vdash \mathcal{O}$* **Gnd**, *$\forall P.\exists Q.\forall R \vdash \mathcal{O}'$* **Slv**, *$\mathcal{Q}\bar{\forall} Q \cup R \vdash \mathcal{O}$* **closed** *and $\mathcal{Q}\bar{\forall} Q \cup R \vdash \mathcal{O}'$* **closed**, *the matching algorithm has the following properties:*

Termination *The appeal $\mathcal{Q}.\forall R \vdash \mathcal{O} \succeq \mathcal{O}' \downarrow \_, \_$ terminates either in success, returning a substitution and realisation, or failure.*

Soundness *If $\mathcal{Q}.\forall R \vdash \mathcal{O} \succeq \mathcal{O}' \downarrow \sigma, \varphi$ then $\mathcal{Q}\bar{\forall} Q \cup R \vdash \sigma$* **valid**, *$\mathrm{Dom}(\varphi) = Q$ and $\sigma(\mathcal{O}) \succeq \varphi\,(\sigma(\mathcal{O}'))$.*

Completeness *If $\sigma_1(\mathcal{O}) \succeq \varphi_1\,(\sigma_1(\mathcal{O}'))$, where $\mathcal{Q}\bar{\forall} Q \cup R \vdash \sigma_1$* **valid**, *$\mathrm{Dom}(\varphi_1) = Q$ and $\mathrm{Reg}(\varphi_1) \cap R = \emptyset$, then, for some σ, φ and σ_2, $\mathcal{Q}.\forall R \vdash \mathcal{O} \succeq \mathcal{O}' \downarrow \sigma, \varphi$ with $\sigma_2 \circ \sigma = \sigma_1$ and $\sigma_2(\varphi) = \varphi_1$.*

8.3.4 A Type Inference Algorithm

With suitable algorithms for unification and matching in hand, we can now present the type inference algorithm for Core-ML with Higher-Order Modules.

Our algorithm has essentially the same form as $\mathcal{W}$, except that it takes an additional argument, the current prefix $\mathcal{Q}$. Intuitively, given a prefix $\mathcal{Q}$, a Higher-Order Modules context $\mathcal{C}$, and a phrase p, the algorithm, if it succeeds, returns a pair consisting of a semantic object o and a $\mathcal{Q}$-substitution σ. If p is a term phrase, then o is the principal classification of the phrase in the most general, inferred context $\sigma(\mathcal{C})$. Similarly, if p is a type phrase, then o is the denotation of the phrase in the most general, inferred context $\sigma(\mathcal{C})$.

To ease the presentation, we shall first define the inference judgements for Core-ML and then for Higher-Order Modules. Although the intended correctness properties of the inference judgements are stated separately, they must, of course, be proven simultaneously, since the judgements are defined by mutual induction.

Type Inference for Core-ML

The inference judgements for Core-ML are designed to satisfy the following property (cf. Theorem 8.5), whose verification is left to future work:

Property 8.17 (Correctness of Core-ML Type Inference).
For any prefix $\mathcal{Q}$ *and context* $\mathcal{C}$ *such that* $\vdash \mathcal{C}$ **Gnd** *and* $\mathcal{Q} \vdash \mathcal{C}$ **closed***:*

Termination:

- *The appeal* $\mathcal{Q}.\mathcal{C} \vdash \mathrm{u} \downarrow \_, \_$ *terminates either in success, returning a simple type and substitution, or failure.*
- *The appeal* $\mathcal{Q}.\mathcal{C} \vdash \mathrm{d} \downarrow \_, \_$ *terminates either in success, returning a definable type and substitution, or failure.*
- *The appeal* $\mathcal{Q}.\mathcal{C} \vdash \mathrm{v} \downarrow \_, \_$ *terminates either in success, returning a value type and substitution, or failure.*
- *The appeal* $\mathcal{Q}.\mathcal{C} \vdash \mathrm{e} \downarrow \_, \_$ *to* monomorphic *type inference terminates either in success, returning a simple type and substitution, or failure.*
- *The appeal* $\mathcal{Q}.\mathcal{C} \vdash \mathrm{e} \downarrow \_, \_$ *to* polymorphic *type inference terminates either in success, returning a value type and substitution, or failure.*

Soundness:

- *If* $\mathcal{Q}.\mathcal{C} \vdash \mathrm{u} \downarrow u, \sigma$*, then* $\mathcal{Q} \vdash \sigma$ **valid** *and* $\sigma(\mathcal{C}) \vdash \mathrm{u} \triangleright u$*.*
- *If* $\mathcal{Q}.\mathcal{C} \vdash \mathrm{d} \downarrow d, \sigma$*, then* $\mathcal{Q} \vdash \sigma$ **valid** *and* $\sigma(\mathcal{C}) \vdash \mathrm{d} \triangleright d$*.*
- *If* $\mathcal{Q}.\mathcal{C} \vdash \mathrm{v} \downarrow v, \sigma$*, then* $\mathcal{Q} \vdash \sigma$ **valid** *and* $\sigma(\mathcal{C}) \vdash \mathrm{v} \triangleright v$*.*

- *If* $\mathcal{Q}.\mathcal{C} \vdash \mathrm{e} \downarrow u, \sigma$*, then* $\mathcal{Q} \vdash \sigma$ **valid** *and* $\sigma(\mathcal{C}) \vdash \mathrm{e} : u$*.*
- *If* $\mathcal{Q}.\mathcal{C} \vdash \mathrm{e} \downarrow v, \sigma$*, then* $\mathcal{Q} \vdash \sigma$ **valid** *and* $\sigma(\mathcal{C}) \vdash \mathrm{e} : v$*.*

Completeness:

- *If* $\sigma_1(\mathcal{C}) \vdash \mathrm{u} \rhd u_1$ *with* $\mathcal{Q} \vdash \sigma_1$ **valid** *then for some* u*,* σ *and* σ_2*,* $\mathcal{Q}.\mathcal{C} \vdash \mathrm{u} \downarrow u, \sigma$ *with* $\sigma_2 \circ \sigma = \sigma_1$ *and* $\sigma_2(u) = u_1$*.*
- *If* $\sigma_1(\mathcal{C}) \vdash \mathrm{d} \rhd d_1$ *with* $\mathcal{Q} \vdash \sigma_1$ **valid** *then for some* d*,* σ *and* σ_2*,* $\mathcal{Q}.\mathcal{C} \vdash \mathrm{d} \downarrow d, \sigma$ *with* $\sigma_2 \circ \sigma = \sigma_1$ *and* $\sigma_2(d) = d_1$*.*
- *If* $\sigma_1(\mathcal{C}) \vdash \mathrm{v} \rhd v_1$ *with* $\mathcal{Q} \vdash \sigma_1$ **valid** *then for some* v*,* σ *and* σ_2*,* $\mathcal{Q}.\mathcal{C} \vdash \mathrm{v} \downarrow v, \sigma$ *with* $\sigma_2 \circ \sigma = \sigma_1$ *and* $\sigma_2(v) = v_1$*.*
- *If* $\sigma_1(\mathcal{C}) \vdash \mathrm{e} : u_1$ *with* $\mathcal{Q} \vdash \sigma_1$ **valid** *then for some* u*,* σ *and* σ_2*,* $\mathcal{Q}.\mathcal{C} \vdash \mathrm{e} \downarrow u, \sigma$ *with* $\sigma_2 \circ \sigma = \sigma_1$ *and* $\sigma_2(u) = u_1$*.*
- *If* $\sigma_1(\mathcal{C}) \vdash \mathrm{e} : v_1$ *with* $\mathcal{Q} \vdash \sigma_1$ **valid** *then for some* v*,* σ *and* σ_2*,* $\mathcal{Q}.\mathcal{C} \vdash \mathrm{e} \downarrow v, \sigma$ *with* $\sigma_2 \circ \sigma = \sigma_1$ *and* $\sigma_2(v) = v_1$*.*[5]

Denotation Inference Rules

The rules for inferring the denotations of Core-ML type phrases are straightforward adaptations of their counterparts in the static semantic of Core-ML (cf. Section 3.2.3). Each rule has been altered to propagate the additional substitution from the output of one recursive call to the inputs of the next, and to return the appropriate denotation and composite substitution, taking care to apply any intervening substitution to a previously computed component of the denotation.

Simple Types $\boxed{\mathcal{Q}.\mathcal{C} \vdash \mathrm{u} \downarrow u, \sigma}$

$$\frac{\mathcal{C}('\mathrm{a}) = u}{\mathcal{Q}.\mathcal{C} \vdash {}'\mathrm{a} \downarrow u, \emptyset} \tag{$\mathcal{C}$-1}$$

$$\frac{\mathcal{Q}.\mathcal{C} \vdash \mathrm{u} \downarrow u, \sigma_1 \quad \sigma_1(\mathcal{Q}).\sigma_1(\mathcal{C}) \vdash \mathrm{u}' \downarrow u', \sigma_2}{\mathcal{Q}.\mathcal{C} \vdash \mathrm{u} \to \mathrm{u}' \downarrow \sigma_2(u) \to u', \sigma_2 \circ \sigma_1} \tag{$\mathcal{C}$-2}$$

[5] The reason that we can state $\sigma_2(v) = v_1$ rather than just $\sigma_2(v) \succeq v_1$, as in the completeness property for algorithm $\mathcal{W}$ (Theorem 8.5), is because the Core-ML judgement $\sigma_1(\mathcal{C}) \vdash \mathrm{e} : v_1$ already requires that the derived type v_1 is *principal* for e in $\sigma_1(\mathcal{C})$ (cf. Rule (C-10)), while the corresponding ML judgement $\sigma_1(\mathcal{C}) \vdash \mathrm{e} : v_1$ does not (cf. Rule (*ML*-6)).

$$\frac{\begin{array}{l}\mathcal{Q}.\mathcal{C} \vdash \mathrm{do} \downarrow d, \sigma \\ \forall i \in [\mathrm{k}].\quad (\sigma_{(i-1)} \circ \ldots \sigma_0 \circ \sigma)(\mathcal{Q}).(\sigma_{(i-1)} \circ \ldots \sigma_0 \circ \sigma)(\mathcal{C}) \vdash \mathrm{u}_i \downarrow u_i, \sigma_i \\ \sigma_{(\mathrm{k}-1)} \circ \ldots \sigma_0(d) \equiv \Lambda('a_0, \ldots, 'a_{\mathrm{k}-1}).u \\ \bar{\sigma} \equiv \{'a_i \mapsto \sigma_{(\mathrm{k}-1)} \circ \ldots \sigma_{(i+1)}(u_i) \mid i \in [\mathrm{k}]\}\end{array}}{\mathcal{Q}.\mathcal{C} \vdash \mathrm{do}(\mathrm{u}_0, \ldots, \mathrm{u}_{\mathrm{k}-1}) \downarrow \bar{\sigma}\ (u), \sigma_{(\mathrm{k}-1)} \circ \ldots \sigma_0 \circ \sigma} \quad (\mathcal{C}\text{-}3)$$

The denotations of definable types and value types are inferred in a similar manner: fresh simple type variables are chosen to represent the bound variables of the phrase. These are declared as *parameters* in the current prefix $\mathcal{Q}$ before inferring the denotation of the body. Declaring the parameters within the scope of all variables in $\mathcal{Q}$ (and, by implication, in $\mathcal{C}$) ensures that these variables remain fresh for the inferred context $\sigma(\mathcal{C})$, respecting the side-conditions of Rules (C-4) and (C-5).

Definable Types

$\boxed{\mathcal{Q}.\mathcal{C} \vdash \mathrm{d} \downarrow d, \sigma}$

$$\frac{\begin{array}{l}\forall i \in [\mathrm{k}].'a_i \notin \mathcal{V}(\mathcal{Q}) \cup \{'a_0, \ldots, 'a_{i-1}\} \\ \mathcal{Q}\bar{\forall}\{'a_0, \ldots, 'a_{\mathrm{k}-1}\}.\mathcal{C}['\mathrm{a}_0 = 'a_0] \cdots ['\mathrm{a}_{\mathrm{k}-1} = 'a_{\mathrm{k}-1}] \vdash \mathrm{u} \downarrow u, \sigma\end{array}}{\mathcal{Q}.\mathcal{C} \vdash \Lambda('\mathrm{a}_0, \cdots, '\mathrm{a}_{\mathrm{k}-1}).\mathrm{u} \downarrow \Lambda('a_0, \cdots, 'a_{\mathrm{k}-1}).u, \sigma} \quad (\mathcal{C}\text{-}4)$$

Value Types

$\boxed{\mathcal{Q}.\mathcal{C} \vdash \mathrm{v} \downarrow v, \sigma}$

$$\frac{\begin{array}{l}\forall i \in [n].'a_i \notin \mathcal{V}(\mathcal{Q}) \cup \{'a_0, \ldots, 'a_{i-1}\} \\ \mathcal{Q}\bar{\forall}\{'a_0, \ldots, 'a_{n-1}\}.\mathcal{C}['\mathrm{a}_0 = 'a_0] \cdots ['\mathrm{a}_{n-1} = 'a_{n-1}] \vdash \mathrm{u} \downarrow u, \sigma\end{array}}{\mathcal{Q}.\mathcal{C} \vdash \forall '\mathrm{a}_0, \cdots, '\mathrm{a}_{n-1}.\mathrm{u} \downarrow \forall 'a_0, \cdots, 'a_{n-1}.u, \sigma} \quad (\mathcal{C}\text{-}5)$$

Classification Inference Rules

The rules for inferring the monomorphic types of Core-ML phrases closely follow the rules of algorithm $\mathcal{W}$. The main difference is this: where our presentation of $\mathcal{W}$ employed the informal notion of generating "fresh" simple type variables, without specifying how fresh is actually fresh enough, we use the current prefix to determine adequate "freshness".

Monomorphic Values

$\boxed{\mathcal{Q}.\mathcal{C} \vdash \mathrm{e} \downarrow u, \sigma}$

$$\frac{\mathcal{C}(\mathrm{i}) = u}{\mathcal{Q}.\mathcal{C} \vdash \mathrm{i} \downarrow u, \emptyset} \quad (\mathcal{C}\text{-}6)$$

$$\frac{{}'a \notin \mathcal{V}(\mathcal{Q}) \quad \mathcal{Q}\exists{}'a.\mathcal{C}[\mathrm{i} : {}'a] \vdash \mathrm{e} \downarrow u', \sigma}{\mathcal{Q}.\mathcal{C} \vdash \lambda\mathrm{i}.\mathrm{e} \downarrow \sigma({}'a) \to u', \sigma \setminus \{{}'a\}} \qquad (\mathcal{C}\text{-}7)$$

$$\frac{\begin{array}{l}\mathcal{Q}.\mathcal{C} \vdash \mathrm{e} \downarrow u, \sigma_1\\ \mathcal{Q}' \equiv \sigma_1(\mathcal{Q})\overline{\exists}(\mathrm{FTVS}(u) \setminus \mathcal{V}(\sigma_1(\mathcal{Q})))\\ \mathcal{Q}'.\sigma_1(\mathcal{C}) \vdash \mathrm{e}' \downarrow u', \sigma_2\\ \mathcal{Q}'' \equiv \sigma_2(\mathcal{Q}')\overline{\exists}(\mathrm{FTVS}(u') \setminus \mathcal{V}(\sigma_2(\mathcal{Q}')))\\ {}'a \notin \mathcal{V}(\mathcal{Q}'')\\ \mathcal{Q}''\exists{}'a \vdash \sigma_2(u) = u' \to {}'a \ \downarrow \ \sigma_3\end{array}}{\mathcal{Q}.\mathcal{C} \vdash \mathrm{e}\ \mathrm{e}' \downarrow \sigma_3({}'a), (\sigma_3 \circ \sigma_2 \circ \sigma_1) \setminus \{{}'a\}} \qquad (\mathcal{C}\text{-}8)$$

$$\frac{\begin{array}{l}\mathcal{Q}.\mathcal{C} \vdash \mathrm{vo} \downarrow \forall{}'a_0, \ldots, {}'a_{n-1}.u, \sigma\\ \forall i \in [n].{}'a_i \notin \mathcal{V}(\sigma(\mathcal{Q})) \cup \{{}'a_0, \ldots, {}'a_{i-1}\}\end{array}}{\mathcal{Q}.\mathcal{C} \vdash \mathrm{vo} \downarrow u, \sigma} \qquad (\mathcal{C}\text{-}9)$$

In Rule ($\mathcal{C}$-7), inferring the type of a function λi.e, we represent the unknown type of the formal argument i as a variable ${}'a$ that is chosen to be fresh for the current prefix $\mathcal{Q}$ (cf. the premise ${}'a$ **fresh** in Rule ($\mathcal{W}$-2)). Before inferring the type of the function body, ${}'a$ is recorded as a meta-variable that is declared within the scope of all the variables in the current prefix $\mathcal{Q}$. In this way, any parameter declared within the scope of ${}'a$, while inferring the type of the body, will be prevented from occurring in the inferred context $\sigma(\mathcal{C}[\mathrm{i} : {}'a]) \equiv \sigma(\mathcal{C})[\mathrm{i} : \sigma({}'a)]$. On the other hand, any parameter that already occurs in the context, and therefore cannot have a side-condition that prevents it from occurring in the additional assumption $\ldots[\mathrm{i} : \sigma({}'a)]$, is free to do so.

A minor complication of our algorithm, that is not apparent in the informal description of $\mathcal{W}$,[6] stems from the need to keep track of any fresh meta-variables that may appear in the inferred type of a subexpression, without having been declared in the prefix used to check that expression. These variables arise as the result of eliminating polymorphism. The generation of fresh meta-variables is particularly evident in Rule ($\mathcal{C}$-9), classifying a value occurrence, but also arises from the inherent polymorphism of λ-abstraction and function application. To illustrate the phenomenon, let's just consider Rule ($\mathcal{C}$-9). Much like Rule ($\mathcal{W}$-4) that eliminates the polymorphism of an identifier in algorithm $\mathcal{W}$, this rule generates new meta-variables by returning a "fresh" instance of the value occurrence's polymorphic type. The

[6]precisely because it is implicit in the informal notion of generating "fresh" variables.

problem is that, although these variables are chosen to be fresh with respect to the current prefix $\mathcal{Q}$, we have no explicit means of recording this fact in the output of the rule. The trick is to leave this book-keeping step not to the rule itself, but to any rule that might have invoked it in a recursive call. Rule ($\mathcal{C}$-8) illustrates the idea. After inferring the pair u and σ_1 for the function e, but *before* inferring the type of the argument e$'$, any simple type variable that occurs free in u but is not already declared in the updated prefix $\sigma_1(\mathcal{Q})$ is assumed to be a "fresh" meta-variable, introduced within the scope of all variables in $\sigma_1(\mathcal{Q})$. These fresh variables are first collected and then declared in the extended prefix $\mathcal{Q}' \equiv \sigma_1(\mathcal{Q})\bar{\exists}(\mathrm{FTVS}(u) \setminus \mathcal{V}(\sigma_1(\mathcal{Q})))$. Only then does the algorithm proceed with the second recursive call, using the updated prefix and the modified context $\sigma_1(\mathcal{C})$ to infer the type u' of the argument e$'$. Similar bookkeeping steps are need to account for any fresh meta-variables in u'. These are recorded in $\mathcal{Q}''$, before continuing with the final appeal to unification. As in Rule ($\mathcal{W}$-3), the appeal to unification ensures that the domain of the function e is equivalent to the type of the argument e$'$. However, unlike Rule ($\mathcal{W}$-3), the most general unifier that equates these types must also respect the side conditions on parameters encoded in the prefix.

Finally, the principal value type of an expression is obtained by quantifying over all those variables that remain "fresh" for the prefix $\sigma(\mathcal{Q})$ in the inferred simple type u:

Polymorphic Values $\boxed{\mathcal{Q}.\mathcal{C} \vdash \mathrm{e} \downarrow v, \sigma}$

$$\frac{\begin{array}{l}\mathcal{Q}.\mathcal{C} \vdash \mathrm{e} \downarrow u, \sigma \\ \{'a_0, \ldots, 'a_{n-1}\} \equiv \mathrm{FTVS}(u) \setminus \mathcal{V}(\sigma(\mathcal{Q}))\end{array}}{\mathcal{Q}.\mathcal{C} \vdash \mathrm{e} \downarrow \forall' a_0, \ldots, 'a_{n-1}.u, \sigma} \qquad (\mathcal{C}\text{-}10)$$

Type Inference for Higher-0rder Modules

The inference judgements for Higher-Order Modules are designed to satisfy the following correctness property, whose verification is left to future work:

Property 8.18 (Correctness of Higher-Order Modules Type Inference).

For any prefix $\mathcal{Q}$ and context $\mathcal{C}$ such that $\vdash \mathcal{C}$ **Gnd** *and $\mathcal{Q} \vdash \mathcal{C}$* **closed***:*

Termination:

- *The appeal $\mathcal{Q}.\mathcal{C} \vdash \mathrm{B} \downarrow \_, \_$ terminates either in success, returning a signature and a substitution, or failure.*

- *The appeal* $\mathcal{Q}.\mathcal{C} \vdash \mathrm{S} \downarrow \_, \_$ *terminates either in success, returning a signature and a substitution, or failure.*
- *The appeal* $\mathcal{Q}.\mathcal{C} \vdash \mathrm{do} \downarrow \_, \_$ *terminates either in success, returning a definable type and a substitution, or failure.*
- *The appeal* $\mathcal{Q}.\mathcal{C} \vdash \mathrm{b} \downarrow \_, \_$ *terminates either in success, returning an existential module type and a substitution, or failure.*
- *The appeal* $\mathcal{Q}.\mathcal{C} \vdash \mathrm{m} \downarrow \_, \_$ *terminates either in success, returning an existential module type and a substitution, or failure.*
- *The appeal* $\mathcal{Q}.\mathcal{C} \vdash \mathrm{vo} \downarrow \_, \_$ *terminates either in success, returning a value type and a substitution, or failure.*

Soundness:

- *If* $\mathcal{Q}.\mathcal{C} \vdash \mathrm{B} \downarrow \mathcal{L},\ \sigma$, *then* $\mathcal{Q} \vdash \sigma$ **valid** *and* $\sigma(\mathcal{C}) \vdash \mathrm{B} \triangleright \mathcal{L}$.
- *If* $\mathcal{Q}.\mathcal{C} \vdash \mathrm{S} \downarrow \mathcal{L}, \sigma$, *then* $\mathcal{Q} \vdash \sigma$ **valid** *and* $\sigma(\mathcal{C}) \vdash \mathrm{S} \triangleright \mathcal{L}$.
- *If* $\mathcal{Q}.\mathcal{C} \vdash \mathrm{do} \downarrow d, \sigma$, *then* $\mathcal{Q} \vdash \sigma$ **valid** *and* $\sigma(\mathcal{C}) \vdash \mathrm{do} \triangleright d$.
- *If* $\mathcal{Q}.\mathcal{C} \vdash \mathrm{b} \downarrow \mathcal{X},\ \sigma$, *then* $\mathcal{Q} \vdash \sigma$ **valid** *and* $\sigma(\mathcal{C}) \vdash \mathrm{b} : \mathcal{X}$.
- *If* $\mathcal{Q}.\mathcal{C} \vdash \mathrm{m} \downarrow \mathcal{X}, \sigma$, *then* $\mathcal{Q} \vdash \sigma$ **valid** *and* $\sigma(\mathcal{C}) \vdash \mathrm{m} : \mathcal{X}$.
- *If* $\mathcal{Q}.\mathcal{C} \vdash \mathrm{vo} \downarrow v, \sigma$, *then* $\mathcal{Q} \vdash \sigma$ **valid** *and* $\sigma(\mathcal{C}) \vdash \mathrm{vo} : v$.

Completeness:

- *If* $\sigma_1(\mathcal{C}) \vdash \mathrm{B} \triangleright \mathcal{L}_1$ *with* $\mathcal{Q} \vdash \sigma_1$ **valid** *then for some* $\mathcal{L}$, σ *and* σ_2, $\mathcal{Q}.\mathcal{C} \vdash \mathrm{B} \downarrow \mathcal{L},\ \sigma$ *with* $\sigma_2 \circ \sigma = \sigma_1$ *and* $\sigma_2(\mathcal{L}) = \mathcal{L}_1$.
- *If* $\sigma_1(\mathcal{C}) \vdash \mathrm{S} \triangleright \mathcal{L}_1$ *with* $\mathcal{Q} \vdash \sigma_1$ **valid** *then for some* $\mathcal{L}$, σ *and* σ_2, $\mathcal{Q}.\mathcal{C} \vdash \mathrm{S} \downarrow \mathcal{L}, \sigma$ *with* $\sigma_2 \circ \sigma = \sigma_1$ *and* $\sigma_2(\mathcal{L}) = \mathcal{L}_1$.
- *If* $\sigma_1(\mathcal{C}) \vdash \mathrm{do} \triangleright d_1$ *with* $\mathcal{Q} \vdash \sigma_1$ **valid** *then for some* d, σ *and* σ_2, $\mathcal{Q}.\mathcal{C} \vdash \mathrm{do} \downarrow \mathcal{L}, \sigma$ *with* $\sigma_2 \circ \sigma = \sigma_1$ *and* $\sigma_2(d) = d_1$.
- *If* $\sigma_1(\mathcal{C}) \vdash \mathrm{b} : \mathcal{X}_1$ *with* $\mathcal{Q} \vdash \sigma_1$ **valid** *then for some* $\mathcal{X}$, σ *and* σ_2, $\mathcal{Q}.\mathcal{C} \vdash \mathrm{b} \downarrow \mathcal{X},\ \sigma$ *with* $\sigma_2 \circ \sigma = \sigma_1$ *and* $\sigma_2(\mathcal{X}) = \mathcal{X}_1$.
- *If* $\sigma_1(\mathcal{C}) \vdash \mathrm{m} : \mathcal{X}_1$ *with* $\mathcal{Q} \vdash \sigma_1$ **valid** *then for some* $\mathcal{X}$, σ *and* σ_2, $\mathcal{Q}.\mathcal{C} \vdash \mathrm{m} \downarrow \mathcal{X}, \sigma$ *with* $\sigma_2 \circ \sigma = \sigma_1$ *and* $\sigma_2(\mathcal{X}) = \mathcal{X}_1$.
- *If* $\sigma_1(\mathcal{C}) \vdash \mathrm{vo} : v_1$ *with* $\mathcal{Q} \vdash \sigma_1$ **valid** *then for some* v, σ *and* σ_2, $\mathcal{Q}.\mathcal{C} \vdash \mathrm{vo} \downarrow v, \sigma$ *with* $\sigma_2 \circ \sigma = \sigma_1$ *and* $\sigma_2(v) = v_1$.

The rules for inferring the denotations and classifications of Modules phrases are straightforward adaptations of their counterparts in the static semantic of Higher-Order Modules (cf. Section 5.5). Each rule has been altered to propagate the additional substitution from the output of one recursive call to the inputs of the next, and to return the appropriate semantic object and composite substitution, taking care to apply any intervening substitution to a previously computed component of the semantic object. Whenever a static semantic rule requires the introduction of type variables that are fresh for the inferred context, the corresponding inference rule simply generates variables that are fresh with respect to the current prefix, records them as parameters declared within the scope of all currently declared variables and then proceeds with type inference. The only inference rules that can make a genuine contribution to the substitution returned, rather than merely propagating the results of recursive calls, are those that require an appeal to the generalised signature matching algorithm, i.e. the rules for functor application (Rule ($\mathcal{H}$-19)), curtailment (Rule ($\mathcal{H}$-20)) and abstraction (Rule ($\mathcal{H}$-21)).

Remark 8.3.1. Observe that the denotation rule for functor signatures (Rule (H-7)) is ambiguous in the sense that it does not uniquely determine the enumeration of the functor's $\forall$-bound type variables, but the choice of enumeration affects the order in which the signature's Λ-bound type variables are *parameterised* by the functor's $\forall$-bound type variables. A similar observation applies to the classification rule for functors (Rule (H-18)): it does not uniquely determine the enumeration of the functor's $\forall$-bound type variables, but the choice of enumeration affects the order in which the functor body's $\exists$-bound abstract types are *skolemised* by the functor's $\forall$-bound type variables. These ambiguities are inconsequential, since the choice of enumeration does not alter the *functional dependencies* captured by the parameterised and skolemised variables. The source of the ambiguity is our design decision that semantic objects $\Lambda P.\mathcal{M}$, $\forall P.\mathcal{M} \rightarrow \mathcal{X}$, and $\exists P.\mathcal{M}$ bind *finite sets* of type variables that, being sets, do not admit fixed enumerations. We could easily remove the ambiguity by replacing the use of *finite sets* of variables in these binding constructs by the use of *finite lists* of variables, so long as we revise the static semantics accordingly. Rather than reformulate the static semantics, we let type inference Rules ($\mathcal{H}$-7) and ($\mathcal{H}$-18) be just as non-deterministic in the enumeration of bound type variables, so that any static semantic derivation that exploits a particular enumeration admits a corresponding type inference derivation that exploits the same enumeration. This allows us to give a simple statement of the completeness of type inference. The type inference rules are deterministic in all other respects.

Denotation Inference Rules

Signature Bodies $\boxed{\mathcal{Q}.\mathcal{C} \vdash \mathrm{B} \downarrow \mathcal{L},\ \sigma}$

$$\frac{\begin{array}{ll}\mathcal{Q}.\mathcal{C} \vdash \mathrm{d} \downarrow d, \sigma_1 & \sigma_1(\mathcal{Q}).(\sigma_1(\mathcal{C}))[\mathrm{t} = d] \vdash \mathrm{B} \downarrow \Lambda P.\mathcal{S},\ \sigma_2 \\ P \cap \mathrm{FV}(\sigma_2(d)) = \emptyset & \mathrm{t} \notin \mathrm{Dom}(\mathcal{S})\end{array}}{\mathcal{Q}.\mathcal{C} \vdash \textbf{type}\ \mathrm{t} = \mathrm{d}; \mathrm{B} \downarrow \Lambda P.\mathrm{t} = \sigma_2(d), \mathcal{S},\ \sigma_2 \circ \sigma_1} \quad (\mathcal{H}\text{-}1)$$

$$\frac{\mathcal{Q}\forall\alpha^{\mathrm{k}}.\mathcal{C}[\mathrm{t} = \alpha^{\kappa}] \vdash \mathrm{B} \downarrow \Lambda P.\mathcal{S},\ \sigma \quad \alpha^{\mathrm{k}} \notin \mathcal{V}(\mathcal{Q}) \cup P \quad \mathrm{t} \notin \mathrm{Dom}(\mathcal{S})}{\mathcal{Q}.\mathcal{C} \vdash \textbf{type}\ \mathrm{t} : \mathrm{k}; \mathrm{B} \downarrow \Lambda\{\alpha^{\mathrm{k}}\} \cup P.\mathrm{t} = \alpha^{\mathrm{k}}, \mathcal{S},\ \sigma} \quad (\mathcal{H}\text{-}2)$$

$$\frac{\begin{array}{ll}\mathcal{Q}.\mathcal{C} \vdash \mathrm{v} \downarrow v, \sigma_1 & \sigma_1(\mathcal{Q}).\sigma_1(\mathcal{C})[\mathrm{x} : v] \vdash \mathrm{B} \downarrow \Lambda P.\mathcal{S},\ \sigma_2 \\ P \cap \mathrm{FV}(\sigma_2(v)) = \emptyset & \mathrm{x} \notin \mathrm{Dom}(\mathcal{S})\end{array}}{\mathcal{Q}.\mathcal{C} \vdash \textbf{val}\ \mathrm{x} : \mathrm{v}; \mathrm{B} \downarrow \Lambda P.\mathrm{x} : \sigma_2(v), \mathcal{S},\ \sigma_2 \circ \sigma_1} \quad (\mathcal{H}\text{-}3)$$

$$\frac{\begin{array}{ll}\mathcal{Q}.\mathcal{C} \vdash \mathrm{S} \downarrow \Lambda P.\mathcal{M}, \sigma_1 & \\ P \cap \mathcal{V}(\sigma_1(\mathcal{Q})) = \emptyset & (\sigma_1(\mathcal{Q})\bar{\forall}P).(\sigma_1(\mathcal{C}))[\mathrm{X} : \mathcal{M}] \vdash \mathrm{B} \downarrow \Lambda P'.\mathcal{S},\ \sigma_2 \\ P' \cap (P \cup \mathrm{FV}(\sigma_2(\mathcal{M}))) = \emptyset & \mathrm{X} \notin \mathrm{Dom}(\mathcal{S})\end{array}}{\mathcal{Q}.\mathcal{C} \vdash \textbf{module}\ \mathrm{X} : \mathrm{S}; \mathrm{B} \downarrow \Lambda P \cup P'.\mathrm{X} : \sigma_2(\mathcal{M}), \mathcal{S},\ \sigma_2 \circ \sigma_1} \quad (\mathcal{H}\text{-}4)$$

$$\frac{}{\mathcal{Q}.\mathcal{C} \vdash \epsilon_{\mathrm{B}} \downarrow \Lambda\emptyset.\epsilon_{\mathcal{S}},\ \emptyset} \quad (\mathcal{H}\text{-}5)$$

Signature Expressions $\boxed{\mathcal{Q}.\mathcal{C} \vdash \mathrm{S} \downarrow \mathcal{L}, \sigma}$

$$\frac{\mathcal{Q}.\mathcal{C} \vdash \mathrm{B} \downarrow \mathcal{L},\ \sigma}{\mathcal{Q}.\mathcal{C} \vdash \textbf{sig}\ \mathrm{B}\ \textbf{end} \downarrow \mathcal{L}, \sigma} \quad (\mathcal{H}\text{-}6)$$

$$\frac{\begin{array}{l}\mathcal{Q}.\mathcal{C} \vdash \mathrm{S} \downarrow \Lambda P.\mathcal{M}, \sigma_1 \\ P \cap \mathcal{V}(\sigma_1(\mathcal{Q})) = \emptyset \quad P = \{\alpha_0^{\kappa_0}, \ldots, \alpha_{n-1}^{\kappa_{n-1}}\} \\ (\sigma_1(\mathcal{C})\bar{\forall}P).(\sigma_1(\mathcal{C}))[\mathrm{X} : \mathcal{M}] \vdash \mathrm{S}' \downarrow \Lambda Q.\mathcal{M}', \sigma_2 \\ Q' \cap (P \cup \mathrm{FV}(\sigma_2(\mathcal{M})) \cup \mathrm{FV}(\Lambda Q.\mathcal{M}')) = \emptyset \\ [Q'/Q] = \{\beta^{\kappa} \mapsto \beta^{\kappa_0 \to \cdots \kappa_{n-1} \to \kappa}\ \alpha_0 \cdots \alpha_{n-1} | \beta^{\kappa} \in Q\} \\ Q' = \{\beta^{\kappa_0 \to \cdots \kappa_{n-1} \to \kappa} | \beta^{\kappa} \in Q\}\end{array}}{\mathcal{Q}.\mathcal{C} \vdash \textbf{funsig}(\mathrm{X}{:}\mathrm{S})\mathrm{S}' \downarrow \Lambda Q'.\forall P.\sigma_2(\mathcal{M}) \to [Q'/Q]\,(\mathcal{M}'), \sigma_2 \circ \sigma_1} \quad (\mathcal{H}\text{-}7)$$

Type Occurrences $\boxed{\mathcal{Q}.\mathcal{C} \vdash \mathrm{do} \downarrow d, \sigma}$

$$\frac{\mathrm{t} \in \mathrm{Dom}(\mathcal{C}) \quad \mathcal{C}(\mathrm{t}) = \tau}{\mathcal{Q}.\mathcal{C} \vdash \mathrm{t} \downarrow \hat{\eta}(\tau), \emptyset} \qquad (\mathcal{H}\text{-}8)$$

$$\frac{\mathcal{Q}.\mathcal{C} \vdash \mathrm{m} \downarrow \exists P.\mathcal{S}, \sigma \quad \mathrm{t} \in \mathrm{Dom}(\mathcal{S}) \quad \mathcal{S}(\mathrm{t}) = \tau \quad P \cap \mathrm{FV}(\tau) = \emptyset}{\mathcal{Q}.\mathcal{C} \vdash \mathrm{m.t} \downarrow \hat{\eta}(\tau), \sigma} \qquad (\mathcal{H}\text{-}9)$$

Classification Inference Rules

Structure Bodies $\boxed{\mathcal{Q}.\mathcal{C} \vdash \mathrm{b} \downarrow \mathcal{X},\ \sigma}$

$$\frac{\begin{array}{ll}\mathcal{Q}.\mathcal{C} \vdash \mathrm{d} \downarrow d, \sigma_1 & \sigma_1(\mathcal{Q}).(\sigma_1(\mathcal{C}))[\mathrm{t} = d] \vdash \mathrm{b} \downarrow \exists P.\mathcal{S},\ \sigma_2 \\ P \cap \mathrm{FV}(\sigma_2(d)) = \emptyset & \mathrm{t} \notin \mathrm{Dom}(\mathcal{S})\end{array}}{\mathcal{Q}.\mathcal{C} \vdash \textbf{type}\ \mathrm{t} = \mathrm{d}; \mathrm{b} \downarrow \exists P.\mathrm{t} = \sigma_2(d), \mathcal{S},\ \sigma_2 \circ \sigma_1} \qquad (\mathcal{H}\text{-}10)$$

$$\frac{\begin{array}{ll}\mathcal{Q}.\mathcal{C} \vdash \mathrm{e} \downarrow v, \sigma_1 & \sigma_1(\mathcal{Q}).(\sigma_1(\mathcal{C}))[\mathrm{x} : v] \vdash \mathrm{b} \downarrow \exists P.\mathcal{S},\ \sigma_2 \\ P \cap \mathrm{FV}(\sigma_2(v)) = \emptyset & \mathrm{x} \notin \mathrm{Dom}(\mathcal{S})\end{array}}{\mathcal{Q}.\mathcal{C} \vdash \textbf{val}\ \mathrm{x} = \mathrm{e}; \mathrm{b} \downarrow \exists P.\mathrm{x} : \sigma_2(v), \mathcal{S},\ \sigma_2 \circ \sigma_1} \qquad (\mathcal{H}\text{-}11)$$

$$\frac{\begin{array}{ll}\mathcal{Q}.\mathcal{C} \vdash \mathrm{m} \downarrow \exists P.\mathcal{M}, \sigma_1 & P \cap \mathcal{V}(\sigma_1(\mathcal{Q})) = \emptyset \\ (\sigma_1(\mathcal{Q})\bar{\forall} P).\sigma_1(\mathcal{C})[\mathrm{X} : \mathcal{M}] \vdash \mathrm{b} \downarrow \exists P'.\mathcal{S},\ \sigma_2 & \\ P' \cap (P \cup \mathrm{FV}(\sigma_2(\mathcal{M}))) = \emptyset & \mathrm{X} \notin \mathrm{Dom}(\mathcal{S})\end{array}}{\mathcal{Q}.\mathcal{C} \vdash \textbf{module}\ \mathrm{X} = \mathrm{m}; \mathrm{b} \downarrow \exists P \cup P'.\mathrm{X} : \sigma_2(\mathcal{M}), \mathcal{S},\ \sigma_2 \circ \sigma_1} \qquad (\mathcal{H}\text{-}12)$$

$$\frac{\begin{array}{ll}\mathcal{Q}.\mathcal{C} \vdash \mathrm{m} \downarrow \exists P.\mathcal{M}, \sigma_1 & P \cap \mathcal{V}(\sigma_1(\mathcal{Q})) = \emptyset \\ (\sigma_1(\mathcal{Q})\bar{\forall} P).\sigma_1(\mathcal{C})[\mathrm{X} : \mathcal{M}] \vdash \mathrm{b} \downarrow \exists P'.\mathcal{S},\ \sigma_2 & P' \cap P = \emptyset\end{array}}{\mathcal{Q}.\mathcal{C} \vdash \textbf{local}\ \mathrm{X}\ =\ \mathrm{m}\ \textbf{in}\ \mathrm{b} \downarrow \exists P \cup P'.\mathcal{S},\ \sigma_2 \circ \sigma_1} \qquad (\mathcal{H}\text{-}13)$$

$$\frac{}{\mathcal{Q}.\mathcal{C} \vdash \epsilon_{\mathrm{b}} \downarrow \exists \emptyset.\epsilon_{\mathcal{S}},\ \emptyset} \qquad (\mathcal{H}\text{-}14)$$

Module Expressions $\boxed{\mathcal{Q}.\mathcal{C} \vdash \mathrm{m} \downarrow \mathcal{X}, \sigma}$

$$\frac{\mathrm{X} \in \mathrm{Dom}(\mathcal{C}) \quad \mathcal{C}(\mathrm{X}) = \mathcal{M}}{\mathcal{Q}.\mathcal{C} \vdash \mathrm{X} \downarrow \exists\emptyset.\mathcal{M}, \emptyset} \tag{$\mathcal{H}$-15}$$

$$\frac{\mathcal{Q}.\mathcal{C} \vdash \mathrm{m} \downarrow \exists P.\mathcal{S}, \sigma \quad \mathrm{X} \in \mathrm{Dom}(\mathcal{S}) \quad \mathcal{S}(\mathrm{X}) = \mathcal{M}}{\mathcal{Q}.\mathcal{C} \vdash \mathrm{m.X} \downarrow \exists P.\mathcal{M}, \sigma} \tag{$\mathcal{H}$-16}$$

$$\frac{\mathcal{Q}.\mathcal{C} \vdash \mathrm{b} \downarrow \mathcal{X},\ \sigma}{\mathcal{Q}.\mathcal{C} \vdash \mathbf{struct}\ \mathrm{b}\ \mathbf{end} \downarrow \mathcal{X}, \sigma} \tag{$\mathcal{H}$-17}$$

$$\frac{\begin{array}{l}
\mathcal{Q}.\mathcal{C} \vdash \mathrm{S} \downarrow \Lambda P.\mathcal{M}, \sigma_1 \\
P \cap \mathcal{V}(\sigma_1(\mathcal{Q})) = \emptyset \quad P = \{\alpha_0^{\kappa_0}, \ldots, \alpha_{n-1}^{\kappa_{n-1}}\} \\
(\sigma_1(\mathcal{Q})\bar{\forall}P).(\sigma_1(\mathcal{C}))[\mathrm{X} : \mathcal{M}] \vdash \mathrm{m} \downarrow \exists Q.\mathcal{M}', \sigma_2 \\
Q' \cap (P \cup \mathrm{FV}(\sigma_2(\mathcal{M})) \cup \mathrm{FV}(\exists Q.\mathcal{M}')) = \emptyset \\
[Q'/Q] = \{\beta^\kappa \mapsto \beta^{\kappa_0 \to \cdots \kappa_{n-1} \to \kappa}\ \alpha_0 \cdots \alpha_{n-1} | \beta^\kappa \in Q\} \\
Q' = \{\beta^{\kappa_0 \to \cdots \kappa_{n-1} \to \kappa} | \beta^\kappa \in Q\}
\end{array}}{\mathcal{Q}.\mathcal{C} \vdash \mathbf{functor}(\mathrm{X} : \mathrm{S})\mathrm{m} \downarrow \exists Q'.\forall P.\sigma_2(\mathcal{M}) \to [Q'/Q]\,(\mathcal{M}'), \sigma_2 \circ \sigma_1} \tag{$\mathcal{H}$-18}$$

$$\frac{\begin{array}{l}
\mathcal{Q}.\mathcal{C} \vdash \mathrm{m} \downarrow \mathcal{X}, \sigma_1 \\
\sigma_1(\mathcal{Q}).\sigma_1(\mathcal{C}) \vdash \mathrm{m}' \downarrow \exists P'.\mathcal{M}'', \sigma_2 \\
\sigma_2(\mathcal{X}) \equiv \exists P.\forall Q.\mathcal{M}' \to \mathcal{M} \\
P \cap \mathcal{V}(\sigma_2 \circ \sigma_1(\mathcal{Q})) = \emptyset \\
P' \cap (\mathcal{V}(\sigma_2 \circ \sigma_1(\mathcal{Q})) \cup P) = \emptyset \\
Q \cap (\mathcal{V}(\sigma_2 \circ \sigma_1(\mathcal{Q})) \cup P \cup P') = \emptyset \\
((\sigma_2 \circ \sigma_1(\mathcal{Q}))\bar{\forall}(P \cup P')).\forall\emptyset \vdash \mathcal{M}'' \succeq \mathcal{M}' \ \downarrow\ \sigma_3, \varphi
\end{array}}{\mathcal{Q}.\mathcal{C} \vdash \mathrm{m}\ \mathrm{m}' \downarrow \exists P \cup P'.\varphi\,(\sigma_3(\mathcal{M})), \sigma_3 \circ \sigma_2 \circ \sigma_1} \tag{$\mathcal{H}$-19}$$

$$\frac{\begin{array}{l}
\mathcal{Q}.\mathcal{C} \vdash \mathrm{m} \downarrow \mathcal{X}, \sigma_1 \\
\sigma_1(\mathcal{Q}).\sigma_1(\mathcal{C}) \vdash \mathrm{S} \downarrow \Lambda P'.\mathcal{M}', \sigma_2 \\
\sigma_2(\mathcal{X}) \equiv \exists P.\mathcal{M} \\
P \cap P' = \emptyset \\
(P \cup P') \cap \mathcal{V}(\sigma_2 \circ \sigma_1(\mathcal{Q})) = \emptyset \\
((\sigma_2 \circ \sigma_1(\mathcal{Q}))\bar{\forall}P).\forall\emptyset \vdash \mathcal{M} \succeq \mathcal{M}' \ \downarrow\ \sigma_3, \varphi
\end{array}}{\mathcal{Q}.\mathcal{C} \vdash \mathrm{m} \succeq \mathrm{S} \downarrow \exists P.\varphi\,(\sigma_3(\mathcal{M}')), \sigma_3 \circ \sigma_2 \circ \sigma_1} \tag{$\mathcal{H}$-20}$$

$$\frac{\begin{array}{l}\mathcal{Q}.\mathcal{C} \vdash \mathrm{m} \downarrow \mathcal{X}, \sigma_1 \\ \sigma_1(\mathcal{Q}).\sigma_1(\mathcal{C}) \vdash \mathrm{S} \downarrow \Lambda P'.\mathcal{M}', \sigma_2 \\ \sigma_2(\mathcal{X}) \equiv \exists P.\mathcal{M} \\ P \cap P' = \emptyset \\ (P \cup P') \cap \mathcal{V}(\sigma_2 \circ \sigma_1(\mathcal{Q})) = \emptyset \\ ((\sigma_2 \circ \sigma_1(\mathcal{Q}))\bar{\forall} P).\forall\emptyset \vdash \mathcal{M} \succeq \mathcal{M}' \ \downarrow \ \sigma_3, \varphi\end{array}}{\mathcal{Q}.\mathcal{C} \vdash \mathrm{m} \setminus \mathrm{S} \downarrow \exists P'.\sigma_3(\mathcal{M}'), \sigma_3 \circ \sigma_2 \circ \sigma_1} \quad (\mathcal{H}\text{-}21)$$

Value Occurrences $\boxed{\mathcal{Q}.\mathcal{C} \vdash \mathrm{vo} \downarrow v, \sigma}$

$$\frac{\mathrm{x} \in \mathrm{Dom}(\mathcal{C}) \quad \mathcal{C}(\mathrm{x}) = v}{\mathcal{Q}.\mathcal{C} \vdash \mathrm{x} \downarrow v, \emptyset} \quad (\mathcal{H}\text{-}22)$$

$$\frac{\mathcal{Q}.\mathcal{C} \vdash \mathrm{m} \downarrow \exists P.\mathcal{S}, \sigma \quad \mathrm{x} \in \mathrm{Dom}(\mathcal{S}) \quad \mathcal{S}(\mathrm{x}) = v \quad P \cap \mathrm{FV}(v) = \emptyset}{\mathcal{Q}.\mathcal{C} \vdash \mathrm{m.x} \downarrow v, \sigma} \quad (\mathcal{H}\text{-}23)$$

8.4 Type Inference for Core-ML with First-Class Modules

Let us briefly consider adapting the algorithms of the preceding section to First-Class Modules, the extension of Core-ML with package types presented in Chapter 7. There are two issues we need to address:

1. the extension of the $\mathcal{Q}$-unification algorithm to deal correctly with the unification of package types $<\exists P.\mathcal{M}> \in \mathit{SimTyp}$, in a way that respects the equivalence in Definition 7.1.

2. the extension of the Core-ML type inference judgements to handle the additional Core-ML phrases for specifying, introducing and eliminating values with package types.

Fortunately, all of the machinery we shall need has already been defined.

8.4.1 Extending the Unification Algorithm

Our unification algorithm must be able to unify simple types up to the equivalence in Definition 7.1. Recall that this definition essentially identifies package types up to re-orderings of components.

Given a unification problem defined by $\mathcal{Q}$, u and u', we need only consider the additional case where either one or both of the simple types is a package type. If only one is a package type, and the other is anything but a meta-variable, then unification must fail. If the other is a meta-variable, then the case is already covered by the algorithm's existing rules (Rule ($\mathcal{U}$-3) or Rule ($\mathcal{U}$-4)).

The only case that remains is the one where u and u' are both package types. Let $u \equiv <\exists P.\mathcal{M}>$ and $u' \equiv <\exists P'.\mathcal{M}'>$. We want to construct the most general $\mathcal{Q}$-substitution σ such that $\sigma(<\exists P.\mathcal{M}>)$ and $\sigma(<\exists P'.\mathcal{M}'>)$ are equal "up to" the equivalence of Definition 7.1.

Intuitively, in the special case where the two package types contain *none* of the meta-variables in $\mathcal{Q}$, then unification reduces to a test for equivalence that, expanding Definition 7.1, can be checked by verifying the conditions:

- $P' \cap \mathrm{FV}(\exists P.\mathcal{M}) = \emptyset$ and $\mathcal{M}' \succeq \varphi\,(\mathcal{M})$ for some φ with $\mathrm{Dom}(\varphi) = P$; and symmetrically:
- $P \cap \mathrm{FV}(\exists P'.\mathcal{M}') = \emptyset$ and $\mathcal{M} \succeq \varphi'\,(\mathcal{M}')$ for some φ' with $\mathrm{Dom}(\varphi') = P'$.

In other words, the two package types are equivalent if, and only if, each is a generic instance of the signature determined by the other. In the sketched proof of Conjecture 7.2, we argued that these conditions could be verified by two symmetric appeals to our original matching algorithm of Chapter 5. (Recall that the well-formedness condition on package types ensures that the corresponding signatures $\Lambda P.\mathcal{M}$ and $\Lambda P'.\mathcal{M}'$ are both solvable ($\vdash \Lambda P.\mathcal{M}$ **Slv** and $\vdash \Lambda P'.\mathcal{M}'$ **Slv**).)

In the more general case, where $<\exists P.\mathcal{M}>$ and $<\exists P'.\mathcal{M}'>$ may contain *meta-variables* declared in $\mathcal{Q}$, this argument suggests that we can perform a test for equivalence modulo unification using our *generalised* matching algorithm. Assuming that $<\exists P.\mathcal{M}>$ and $<\exists P'.\mathcal{M}'>$ are $\mathcal{Q}$-closed, we want to obtain a most general $\mathcal{Q}$-substitution σ such that $\sigma(<\exists P.\mathcal{M}>)$ and $\sigma(<\exists P'.\mathcal{M}'>)$ are equivalent. Without loss of generality we may also assume that $(P \cup P') \cap \mathcal{V}(\mathcal{Q}) = \emptyset$ and $P \cap P' = \emptyset$. Given that $<\exists P.\mathcal{M}>$ and $<\exists P'.\mathcal{M}'>$ are $\mathcal{Q}$-closed, these conditions ensure that $P' \cap \mathrm{FV}(\exists P.\mathcal{M}) = \emptyset$ and $P \cap \mathrm{FV}(\exists P'.\mathcal{M}') = \emptyset$. In the first appeal to the generalised matching algorithm, we treat variables in P' as generic parameters and determine the most general $\mathcal{Q}\bar{\forall}(P' \cup P)$-substitution σ_1 and realisation φ such that $\sigma_1(\mathcal{M}') \succeq \varphi\,(\sigma_1(\mathcal{M}))$ with $\mathrm{Dom}(\varphi) = P$. In a second appeal, we determine the most general $\sigma_1(\mathcal{Q})\bar{\forall}(P \cup P')$-substitution σ_2 and realisation φ', such that $\sigma_2(\sigma_1(\mathcal{M})) \succeq \varphi'\,(\sigma_2(\sigma_1(\mathcal{M}')))$ with $\mathrm{Dom}(\varphi') = P'$.

Since enrichment is closed under substitution, applying σ_2 to the first relation we obtain $\sigma_2(\sigma_1(\mathcal{M}')) \succeq \sigma_2(\varphi\,(\sigma_1(\mathcal{M})))$. Since σ_2 is a $\sigma_1(\mathcal{Q})\bar{\forall}(P \cup P')$-substitution and $\sigma_1(\mathcal{M})$ is $\sigma_1(\mathcal{Q})\bar{\forall}(P \cup P')$-closed, we can express the relation $\sigma_2(\sigma_1(\mathcal{M}')) \succeq \sigma_2(\varphi\,(\sigma_1(\mathcal{M})))$ as $\sigma_2 \circ \sigma_1(\mathcal{M}') \succeq \varphi''\,(\sigma_2 \circ \sigma_1(\mathcal{M}))$, where φ'' is the realisation $\varphi'' \stackrel{\text{def}}{=} \sigma_2(\varphi)$. By the definition of $\sigma_2 \circ \sigma_1$, $\sigma_2(\sigma_1(\mathcal{M})) \succeq \varphi'\,(\sigma_2(\sigma_1(\mathcal{M}')))$ can be re-expressed as $\sigma_2 \circ \sigma_1(\mathcal{M}) \succeq \varphi'\,(\sigma_2 \circ \sigma_1(\mathcal{M}'))$. We can then verify the conditions of Definition 7.1 to show that $<\exists P.\sigma_2 \circ \sigma_1(\mathcal{M})>$ is equivalent to $<\exists P'.\sigma_2 \circ \sigma_1(\mathcal{M}')>$. Since the parameters in P and P' cannot occur in the images of σ_1 and σ_2, it follows that $\sigma_2 \circ \sigma_1(<\exists P.\mathcal{M}>) = \sigma_2 \circ \sigma_1(<\exists P'.\mathcal{M}'>)$, i.e. that $\sigma_2 \circ \sigma_1$ is a $\mathcal{Q}$-unifier of u and u'. Furthermore, it is not difficult to reason that $\sigma_2 \circ \sigma_1$ is a most general $\mathcal{Q}$-unifier of u and u'; and that if either of the above appeals to matching fails, then u and u' do not have a $\mathcal{Q}$-unifier.

This idea is captured by extending the unification algorithm with the following rule:

$$\boxed{\mathcal{Q} \vdash u = u' \downarrow \sigma}$$

$$\frac{\begin{array}{l}(P \cup P') \cap \mathcal{V}(\mathcal{Q}) = \emptyset \\ P \cap P' = \emptyset \\ \mathcal{Q}\bar{\forall}P'.\forall\emptyset \vdash \mathcal{M}' \succeq \mathcal{M} \downarrow \sigma_1, \varphi \\ (\sigma_1(\mathcal{Q}))\bar{\forall}P.\forall\emptyset \vdash \sigma_1(\mathcal{M}) \succeq \sigma_1(\mathcal{M}') \downarrow \sigma_2, \varphi'\end{array}}{\mathcal{Q} \vdash <\exists P.\mathcal{M}> = <\exists P'.\mathcal{M}'> \downarrow \sigma_2 \circ \sigma_1} \qquad (\mathcal{H}\text{-}24)$$

Remark 8.4.1. Clearly, adding this rule means that the proofs of termination, soundness and completeness of our unification and matching algorithms must now be carried out simultaneously. Even though it is easy to motivate and describe the extension, it is by no means obvious that the properties of the original, stratified, algorithms are preserved: we have tied the knot and made them mutually recursive.

For instance, one of the challenges is to devise a decreasing measure on unification problems that establishes termination. The measure traditionally used for first-order unification relies on the fact that the unification algorithm cannot introduce new meta-variables during its execution. This property no longer holds in the extended algorithm since it may indirectly invoke Rule ($\mathcal{V}$-1) and, in Example 8.3.4, we demonstrated how this rule can increase the set of new meta-variables.

8.4.2 Extending the Type Inference Algorithm

It remains to extend the inference judgements of Core-ML with the three additional inference rules dealing with package types. Rules ($\mathcal{P}$-1), ($\mathcal{P}$-2) and ($\mathcal{P}$-3) are straightforward adaptations of the corresponding static semantic rules, Rules (P-1), (P-2) and (P-3):

Denotation Inference Rules $\boxed{\mathcal{Q}.\mathcal{C} \vdash \mathrm{u} \downarrow u, \sigma}$

$$\frac{\mathcal{Q}.\mathcal{C} \vdash \mathrm{S} \downarrow \Lambda P.\mathcal{M}, \sigma}{\mathcal{Q}.\mathcal{C} \vdash \texttt{<}\mathrm{S}\texttt{>} \downarrow \texttt{<}\exists P.\mathcal{M}\texttt{>}, \sigma} \qquad (\mathcal{P}\text{-}1)$$

Inferring the denotation of the type phrase <S> is easy: we simply infer the denotation of the signature S, derive the corresponding package type and propagate the substitution from the recursive call.

Classification Inference Rules $\boxed{\mathcal{Q}.\mathcal{C} \vdash \mathrm{e} \downarrow u, \sigma}$

$$\frac{\begin{array}{l} \mathcal{Q}.\mathcal{C} \vdash \mathrm{m} \downarrow \mathcal{X}, \sigma_1 \\ \sigma_1(\mathcal{Q}).\sigma_1(\mathcal{C}) \vdash \mathrm{S} \downarrow \Lambda P.\mathcal{M}, \sigma_2 \\ \sigma_2(\mathcal{X}) \equiv \exists P'.\mathcal{M}' \\ P' \cap \mathcal{V}(\sigma_2 \circ \sigma_1(\mathcal{Q})) = \emptyset \\ P \cap (\mathcal{V}(\sigma_2 \circ \sigma_1(\mathcal{Q})) \cup P') = \emptyset \\ ((\sigma_2 \circ \sigma_1(\mathcal{Q}))\bar{\forall} P').\forall\emptyset \vdash \mathcal{M}' \succeq \mathcal{M} \ \downarrow \ \sigma_3, \varphi \end{array}}{\mathcal{Q}.\mathcal{C} \vdash \textbf{pack}\ \mathrm{m}\ \textbf{as}\ \mathrm{S} \downarrow \texttt{<}\exists P.\sigma_3(\mathcal{M})\texttt{>}, \sigma_3 \circ \sigma_2 \circ \sigma_1} \qquad (\mathcal{P}\text{-}2)$$

Inferring the classification of the phrase **pack** m **as** S requires an appeal to the matching algorithm, and is similar to inferring the classification of an abstraction (Rule ($\mathcal{H}$-21)).

$$\frac{\begin{array}{l} \mathcal{Q}.\mathcal{C} \vdash \mathrm{e} \downarrow u, \sigma_1 \\ \mathcal{Q}' \equiv \sigma_1(\mathcal{Q})\bar{\exists}(\mathrm{FTVS}(u) \setminus \mathcal{V}(\sigma_1(\mathcal{Q}))) \\ \mathcal{Q}'.\sigma_1(\mathcal{C}) \vdash \mathrm{S} \downarrow \Lambda P.\mathcal{M}, \sigma_2 \\ \sigma_2(\mathcal{Q}') \vdash u = \texttt{<}\exists P.\mathcal{M}\texttt{>} \ \downarrow \ \sigma_3 \\ \sigma_3(\texttt{<}\exists P.\mathcal{M}\texttt{>}) \equiv \texttt{<}\exists P'.\mathcal{M}'\texttt{>} \quad P' \cap \mathcal{V}(\sigma_3 \circ \sigma_2(\mathcal{Q}')) = \emptyset \\ (\sigma_3 \circ \sigma_2(\mathcal{Q}')\bar{\forall} P').(\sigma_3 \circ \sigma_2 \circ \sigma_1)(\mathcal{C})[\mathrm{X} : \mathcal{M}'] \vdash \mathrm{e}' \downarrow u', \sigma_4 \\ P' \cap \mathrm{FV}(u') = \emptyset \end{array}}{\mathcal{Q}.\mathcal{C} \vdash \textbf{open}\ \mathrm{e}\ \textbf{as}\ \mathrm{X} : \mathrm{S}\ \textbf{in}\ \mathrm{e}' \downarrow u', \sigma_4 \circ \sigma_3 \circ \sigma_2 \circ \sigma_1} \qquad (\mathcal{P}\text{-}3)$$

In Rule ($\mathcal{P}$-3), we can observe that the presence of the explicit signature in the elimination phrase **open** e **as** X : S **in** e′ means that the simple type of e is uniquely determined up to unifiability with $<\exists P.\mathcal{M}>$. Since this is the only phrase that eliminates package types, the type inference algorithm never needs to make uneducated guesses about the encapsulated, higher-order type structure of value expressions that are actually used as packages. We conjecture that this explicitly typed elimination construct preserves the principal typing property, i.e. that every typable expression has a principal type.

8.5 Conclusion

In this chapter, we took a detailed look at the problem of combining type inference for Core-ML and type checking of Higher-Order Modules. After reviewing ML type inference, we gave counter-examples that illustrated how the naive approach of interleaving ML's type inference algorithm $\mathcal{W}$ with the Modules type checker of Chapter 5 fails to be sound. We used these examples to motivate the design of an integrated algorithm, similar in spirit to $\mathcal{W}$ but, by necessity, more complex. Although we stated the key correctness properties that the algorithms are designed to satisfy, the proof of these properties must be left to future work. The main obstacle to proving these properties is simply the sheer number of cases to consider.

To alleviate this deficiency, we have aimed for a clear presentation of the concepts underlying the algorithms, building on the more solid foundations of Chapter 5 and taking inspiration from the well-known properties of ML and its type inference algorithm. This leaves us in the unsatisfactory but probably unavoidable situation of relying on our engineering judgement to assess the correctness of the algorithms.

However, empirical evidence does support our claim of correctness. The algorithms, including the extension to First-Class Modules, are implemented in the prototype interpreter accompanying this thesis [Rus98a]. The algorithms behave correctly on a small but representative range of tests that includes all of the examples in this thesis.

Chapter 9

Conclusion

We conclude this thesis with a summary of our achievements (Section 9.1), a comparison with related research (Section 9.2), and directions for further work (Section 9.3).

9.1 Summary

The Standard ML Modules language has been both the subject and the source of much of the recent research into the type-theoretic foundations of module languages. Despite these efforts, a proper type-theoretic understanding of the static semantics of Modules has not emerged. Such an understanding offers two potential benefits: Type Theory provides us with a rational basis for analysing existing features of the language, and for synthesising new features by generalisation.

In Chapter 2, we reviewed how the existing type-theoretic accounts of Standard ML, based on a syntactic reduction to standard constructs from Type Theory, are largely unsatisfactory. The more successful type-theoretic alternatives to Standard ML Modules (cf. Section 2.3.3), although extending its capabilities considerably, have resorted to introducing non-standard constructs with unpleasant meta-theoretic properties. We consequently could discern no distinct advantage in abandoning the existing semantics of Standard ML, provided its main aspects could be explained and extended directly by analogy with Type Theory.

Thus we undertook the work in this thesis with two main objectives. The first was to provide a better, more type-theoretic formulation of the existing static semantics of Modules. The second was to use this formulation as the rational basis for designing proper extensions of Modules.

In Chapter 3 we gave a stylised presentation of the existing static semantics of Standard ML. We took great pains to separate the semantics of the Core and Modules, distilling the essence of Modules and making precise the extent to which each of Core and Modules depends on the other. This effort was motivated by the desire to ensure that both the syntax and semantics of Modules were more amenable to generalisation, and to permit applications of Modules to different Core languages. Examining the semantics we found no evidence to support the often made claim that a type-theoretic model of Modules requires first-order dependent types. Aside from the operational use of the state of type variables to implement type generativity, the type structure of Modules is easily explained by resorting to the simpler, second-order notions of type parameterisation (for signatures), universal quantification over types (for functors), and subtyping (for enrichment).

In Chapter 4 we presented a new, more declarative static semantics for Modules, based on classifying structures using existentially quantified types. Our main objective was to first explain and then eliminate the state of type variables maintained by the generative classification judgements of Chapter 3. By proving the equivalence of the two semantics, we showed that type generativity is nothing more than a procedural implementation of existential quantification over types. In presenting the new semantics, we also adopted some clarifying notational changes, stressing the role of signatures as parameterised types, and functors as polymorphic functions.

In Chapter 5, we extended the Modules language of Chapter 3 to higher-order, using the revised semantics in Chapter 4 as our starting point. Functors were given the status previously enjoyed only by structures: they could now be defined as components of structures, specified as functor arguments and returned as functor results. We were able to present the semantics of Higher-Order Modules as a natural generalisation of the definitions underlying the first-order language of the preceding chapters. The crucial ideas of introducing higher-order realisations and of generalising the enrichment relation to functors, by combining polymorphic subsumption with contravariant enrichment, were adapted and reworked from the original proposals of Biswas [Bis95]. The applicative semantics for functor generativity was inspired by our own results in Chapter 4. We also addressed the practical concern of type-checking Modules by providing, and proving correct, a sound and complete algorithm for signature matching. The algorithm is similar to, but simpler than, the one proposed by Biswas.

In Chapter 6 we briefly discussed the foundations of a separate compilation system for Modules. Using our revised semantics, we were able to analyse why the traditional approach to separate compilation in Standard ML fails. We managed to identify an alternative notion of compilation unit that satisfies the requirements of separate compilation. Although acceptable in practice, from a theoretical perspective this solution is only partial. After analysing the problem, we suggested appropriate modifications to the syntax and semantics, and formalised these ideas in a skeletal higher-order modules calculus. We sketched a proof of the adequacy of these modifications.

In Chapter 7 we turned our attention to a particular Core language, Core-ML, and relaxed the stratification between Core and Modules. We obtained a language with first-class modules and gave examples of programs exploiting them. Our approach is novel in maintaining the distinction between Core and Modules. Instead of amalgamating the features of both in a *single* language, we provide constructs for packing Module values as Core values and opening Core values as Module values, allowing programs to alternate between Modules and Core level computation. Our ability to define a simple notion of first-class module directly contradicts the claims made by Harper and Mitchell [HM93]: their analysis implies that Standard ML is incompatible with first-class modules.

In Chapter 8 we considered the type inference problem posed by Core-ML in the presence of both higher-order and first-class modules. We reviewed the classical, unification-based type inference algorithm for ML, the language on which Core-ML is founded. We discussed why a naive combination of the type checker for Modules with the traditional type inference algorithm for ML is inadequate. We designed a suitably generalised unification algorithm, and presented a derived, hybrid type inference algorithm that integrates type checking of Modules with type inference for Core-ML. We stated correctness properties of these algorithms but left their verification to future work.

Throughout this thesis, our approach has been to use concepts from type theory as a guideline for reformulating and generalising the *existing* semantics of Modules. An important practical benefit of this approach is that our extensions to the language can readily be integrated with the existing definition and implementations of Standard ML.

9.2 Comparison with Related Work

9.2.1 The Adequacy of MacQueen's Type-Theoretic Analogy

The existing type-theoretic accounts of Standard ML Modules are rooted in an informal analogy, due originally to MacQueen, relating structures to nested pairs, signatures to dependent products, functors to functions, and functor signatures to dependent function spaces. This analogy was first expressed in MacQueen's language DL [Mac86], elaborated in Harper and Mitchell's XML [HM93], and ultimately refined to account for the phase distinction in Harper, Mitchell and Moggi's HML [HMM90]. Certain aspects of this analogy, in particular the reliance on first-order dependent types, can still be found in the type-theoretic alternatives to Standard ML, namely Leroy's Modules [Ler94, Ler96b, Ler95], Harper and Lillibridge's translucent sums calculus [HL94] and its descendants [Lil97, SH96, HS97].

Having familiarised ourselves with the static semantics of Modules, we are now in a good position to assess the adequacy of MacQueen's analogy. Fundamental to MacQueen's argument is the tenet that signatures are the *types* of structures. In MacQueen's interpretation, the specification of a type component within a signature binds an existentially quantified type variable, while the definition of a type component within a structure introduces an existential quantifier. This is consistent with MacQueen's tenet, since it means that the type of a structure is indeed a signature. To account for the transparency of type definitions, MacQueen argues that the existential quantifier must have the strong interpretation of Section 2.2.6. In this way, Standard ML's ability to project the actual type component from a structure is interpreted as the ability to project the type witness from a term of strong existential type, i.e. the dot-notation is modeled by existential elimination.

Remark 9.2.1. Before we start with our examples, we should point out that MacQueen uses pairing instead of naming to group components into structures. For the examples of this section, we shall ignore this discrepancy, since it isn't central to the argument.

Example 9.2.1. Let's illustrate MacQueen's analogy with an example. We will use the informal notation $[\![\mathrm{p}]\!]$ to denote MacQueen's interpretation of the Modules phrase p. For instance, the signature:

$$\mathrm{S} \overset{\text{def}}{=} \textbf{sig type t} : 0; \textbf{val x} : \textbf{t end}$$

is interpreted as the strong existential type:

$$[\![\mathrm{S}]\!] \overset{\text{def}}{=} \Sigma\alpha{:}0.\alpha$$

According to the static semantics of Modules, the denotation of the signature S is:

$$\vdash \mathrm{S} \triangleright \Lambda\{\alpha\}.(\mathbf{t} = \alpha, \mathbf{x} : \alpha).$$

Let's define $\mathcal{L}$ to abbreviate this semantic signature:

$$\mathcal{L} \stackrel{\text{def}}{=} \Lambda\{\alpha\}.(\mathbf{t} = \alpha, \mathbf{x} : \alpha).$$

Let's compare S's denotation $\mathcal{L}$ with MacQueen's interpretation $[\![\mathrm{S}]\!]$. From $\mathcal{L}$ we can see that the function of the type specification in S is *not* to declare a quantified type component: instead, it simultaneously introduces a new type *parameter* α, and declares a type component $\mathbf{t} = \alpha$ denoting this parameter. Notice that the denotation of this type component, although represented by a formal type variable, is explicit in the body $(\mathbf{t} = \alpha, \mathbf{x} : \alpha)$ of $\mathcal{L}$. On the other hand, in the type $[\![\mathrm{S}]\!]$ the existential *quantifier* ensures that the denotation of the type component is *hidden*: $[\![\mathrm{S}]\!]$ tells us only the first component of a pair of type $[\![\mathrm{S}]\!]$ is *some* type of kind 0, without revealing which type.

Despite these differences, let's continue to develop MacQueen's analogy. The structure:

$$\mathrm{s} \stackrel{\text{def}}{=} \textbf{struct type t} = \textbf{int}; \textbf{val x} = 1 \textbf{ end}$$

is interpreted as pairing the type **int** with the term 1 to introduce a value of existential type, for instance:

$$[\![\mathrm{s}]\!] \stackrel{\text{def}}{=} \langle \textbf{int}, 1\rangle \textbf{ as } \Sigma\alpha{:}0.\alpha.$$

Notice that the type of this pair is indeed the existential type $[\![\mathrm{S}]\!]$, since

$$\vdash \langle \textbf{int}, 1\rangle \textbf{ as } \Sigma\alpha{:}0.\alpha : \Sigma\alpha{:}0.\alpha$$

we have:

$$\vdash [\![\mathrm{s}]\!] : [\![\mathrm{S}]\!],$$

which is consistent with MacQueen's interpretation of signatures as types.

According to the static semantics of Modules, the semantic object of the structure s is:

$$\vdash \mathrm{s} : \exists\emptyset.(\mathbf{t} = \textbf{int}, \mathbf{x} : \textbf{int}).$$

Let's define $\mathcal{X}$ to abbreviate the semantic object of s:

$$\mathcal{X} \stackrel{\text{def}}{=} \exists\emptyset.(\mathbf{t} = \textbf{int}, \mathbf{x} : \textbf{int})$$

Now let's compare the type $[\![\mathrm{S}]\!]$ of $[\![\mathrm{s}]\!]$ with the semantic object $\mathcal{X}$ of the structure s. The empty existential quantifier of $\mathcal{X}$ is irrelevant. What is pertinent is that the denotation of the type component, i.e. **int**, is apparent in the semantic object $\mathcal{X}$, while it is hidden in the type $[\![\mathrm{S}]\!]$. This distinction is crucial because it means that the denotation of s's type component can be determined by simple inspection of the semantic object $\mathcal{X}$:

$$\frac{\begin{array}{c}\vdash \mathrm{s} : \exists\emptyset.(\mathbf{t} = \mathbf{int}, \mathbf{x} : \mathbf{int}) \\ \mathbf{t} \in \mathrm{Dom}(\mathbf{t} = \mathbf{int}, \mathbf{x} : \mathbf{int}) \\ (\mathbf{t} = \mathbf{int}, \mathbf{x} : \mathbf{int})(\mathbf{t}) = \mathbf{int}\end{array}}{\vdash \mathrm{s}.\mathbf{t} \triangleright \mathbf{int}}$$

On the other hand, to determine the denotation of $[\![\mathrm{s}]\!]$'s type component, accessed using the first projection $\mathbf{Fst}\ [\![s]\!]$, we have to inspect the term $[\![\mathrm{s}]\!]$ itself, not just its type $[\![\mathrm{S}]\!]$. Only by expanding the definition of $[\![\mathrm{s}]\!]$ can we derive:

$$\frac{\vdash \mathbf{Fst}\ (\langle \mathbf{int}, 1\rangle\ \mathbf{as}\ \Sigma\alpha{:}0.\alpha) : 0}{\vdash \mathbf{Fst}\ (\langle \mathbf{int}, 1\rangle\ \mathbf{as}\ \Sigma\alpha{:}0.\alpha) = \mathbf{int} : 0}$$

and hence:

$$\vdash \mathbf{Fst}\ ([\![\mathrm{s}]\!]) = \mathbf{int} : 0.$$

At first, this difference seems insignificant, since $\mathcal{X}$ must also be obtained by "inspecting" (i.e. classifying) s. However, suppose the structure expression s is not in the canonical form of a structure body, but, for the sake of argument, a functor application. Then we can still determine its semantic object and the denotation of its type component *statically* without dynamically reducing the functor application. In MacQueen's approach, the functor application in s is interpreted as a corresponding function application in $[\![\mathrm{s}]\!]$. Now, because the function application $[\![\mathrm{s}]\!]$ is not in the canonical form of a pair, the application must first be *dynamically* reduced to a pair to discover its type component. Although the *effect* of having a transparent type component is similar in both cases, the mechanisms used to achieve this effect are clearly completely different: static typing on the one hand, dynamic reduction on the other.

MacQueen's interpretation of functors as functions is an extension of the interpretation of signatures as types: because the argument signature of a functor is interpreted as a type, it makes sense that the functor itself be modeled as a function on elements of this type. The need to type functions using first-order universal quantification, i.e. dependent function spaces, follows from the fact that the argument signature may contain existentially quantified, and thus opaque, type components. To see why, consider a function defined with respect to a formal argument whose type is a strong existential. The type of the function's body may mention the argument's type component, expressed as a projection from the argument. However, because the argument is merely formal, i.e. a variable, it cannot be reduced to a pair and the dependency of the function's range on its argument cannot be eliminated. The only way to account for the dependency in the function's type is to use first-order universal quantification of the function's argument over its range.

Example 9.2.2. Continuing Example 9.2.1, consider the functor:

$$\begin{array}{lll} \mathrm{F} & \overset{\mathrm{def}}{=} & \mathbf{functor}(\mathbf{X} : \mathrm{S})\mathbf{struct}\ \mathbf{val}\ \mathbf{y} = \mathbf{X}.\mathbf{x}\ \mathbf{end} \\ & \equiv & \mathbf{functor}(\mathbf{X} : \mathbf{sig}\ \mathbf{type}\ \mathbf{t} : 0; \mathbf{val}\ \mathbf{x} : \mathbf{t}\ \mathbf{end})\mathbf{struct}\ \mathbf{val}\ \mathbf{y} = \mathbf{X}.\mathbf{x}\ \mathbf{end}. \end{array}$$

In MacQueen's interpretation, this corresponds to a λ-abstraction, taking a term of existential type as an argument, and projecting its second component.

$$\begin{array}{lll} [\![\mathrm{F}]\!] & \overset{\mathrm{def}}{=} & \lambda \mathrm{X}{:}[\![\mathrm{S}]\!].\mathbf{Snd}\ \mathrm{X} \\ & \equiv & \lambda \mathrm{X}{:}(\Sigma\alpha{:}0.\alpha).\mathbf{Snd}\ \mathrm{X}. \end{array}$$

The body of this function has type:

$$\mathrm{X} : \Sigma\alpha{:}0.\alpha \vdash \mathbf{Snd}\ \mathrm{X} : \mathbf{Fst}\ \mathrm{X}$$

Moreover, since X is canonical, but *not* in the form of a pair, the type **Fst** X may not be simplified any further. Hence the dependency of the body's type on the *term* X cannot be removed and the function must be given the *first-order* universally quantified type:

$$\vdash \lambda \mathrm{X}{:}(\Sigma\alpha{:}0.\alpha).\mathbf{Snd}\ \mathrm{X} : \forall \mathrm{X}{:}(\Sigma\alpha{:}0.\alpha).\mathbf{Fst}\ \mathrm{X},$$

i.e.

$$\vdash [\![\mathrm{F}]\!] : \forall \mathrm{X}{:}[\![\mathrm{S}]\!].\mathbf{Fst}\ \mathrm{X}.$$

Let's compare the type of $[\![\mathrm{F}]\!]$ with the semantic object that is assigned to the functor F according to the static semantics of Modules:

$$\vdash \mathrm{F} : \exists\emptyset.\forall\{\alpha\}.(\mathbf{t} = \alpha, \mathbf{x} : \alpha) \to (\mathbf{y} : \alpha).$$

Again, the existential quantification is irrelevant. What is pertinent is that the range of the functor shows no first-order dependency on the functor argument: instead, it has a second-order dependency on a universally quantified type variable. What MacQueen interprets as a dependent function on a dependent domain, can be understood as a *polymorphic*, *non-dependent* function on a *non-dependent* domain.

It is revealing to compare the derivation of the function's type:

$$\frac{\mathrm{X}{:}[\![\mathrm{S}]\!] \vdash \mathbf{Snd}\ \mathrm{X} : \mathbf{Fst}\ \mathrm{X}}{\vdash \lambda \mathrm{X}{:}[\![\mathrm{S}]\!].\mathbf{Snd}\ \mathrm{X} : \forall \mathrm{X}{:}[\![\mathrm{S}]\!].\mathbf{Fst}\ \mathrm{X}}$$

with the derivation of the functor's semantic object:

$$\frac{\begin{array}{l}\vdash \mathrm{S} \rhd \Lambda\{\alpha\}.(\mathbf{t} = \alpha, \mathbf{x} : \alpha) \\ \mathbf{X} : (\mathbf{t} = \alpha, \mathbf{x} : \alpha) \vdash \textbf{struct val y} = \mathbf{X}.\mathbf{x}\ \textbf{end} : \exists\emptyset.(\mathbf{y} : \alpha)\end{array}}{\begin{array}{c}\vdash \textbf{functor}(\mathbf{X} : \mathrm{S})\textbf{struct val y} = \mathbf{X}.\mathbf{x}\ \textbf{end} : \\ \exists\emptyset.\forall\{\alpha\}.(\mathbf{t} = \alpha, \mathbf{x} : \alpha) \to (\mathbf{y} : \alpha)\end{array}}$$

Contrast the classification of the function body with the classification of the functor body. Unlike the specified type $[\![\mathrm{S}]\!]$ of the function argument, neither the signature S, nor its denotation $\Lambda\{\alpha\}.(\mathbf{t} = \alpha, \mathbf{x} : \alpha)$, is used directly as the type of the functor argument. Instead, the signature is used in an ancillary role, to specify the family of argument types $\Lambda\{\alpha\}.(\mathbf{t} = \alpha, \mathbf{x} : \alpha)$, indexed by α. The functor's term argument X is assumed to have the type $(\mathbf{t} = \alpha, \mathbf{x} : \alpha)$, which is a generic member of this family. Intuitively, discharging the functor's term parameter X yields a *monomorphic* function of type $(\mathbf{t} = \alpha, \mathbf{x} : \alpha) \to (\mathbf{y} : \alpha)$. Discharging the functor's type parameter α yields a *polymorphic* function of type $\forall\{\alpha\}.(\mathbf{t} = \alpha, \mathbf{x} : \alpha) \to (\mathbf{y} : \alpha)$. The derivation combines both these steps into one.

We can also see the difference between dependent functions and functors by comparing how they are applied. For example, the type of the application of the function $[\![\mathrm{F}]\!]$ to the actual argument $[\![\mathrm{s}]\!]$ is obtained by a *first-order* substitution, substituting the term $[\![\mathrm{s}]\!]$ for the quantified term variable X in the range of $[\![\mathrm{F}]\!]$:

$$\frac{\vdash [\![\mathrm{F}]\!] : \forall \mathrm{X}{:}[\![\mathrm{S}]\!].\mathbf{Fst}\ \mathrm{X} \qquad \vdash [\![\mathrm{s}]\!] : [\![\mathrm{S}]\!]}{\vdash [\![\mathrm{F}]\!]\ [\![\mathrm{s}]\!] : [[\![\mathrm{s}]\!]/\mathrm{X}]\,(\mathbf{Fst}\ \mathrm{X}).}$$

On the other hand, consider the classification of the corresponding functor application according to the static semantics of Modules:

$$\frac{\begin{array}{l}\vdash \mathrm{F} : \exists\emptyset.\forall\{\alpha\}.(\mathbf{t} = \alpha, \mathbf{x} : \alpha) \to (\mathbf{y} : \alpha) \\ \vdash \mathrm{s} : \exists\emptyset.(\mathbf{t} = \mathbf{int}, \mathbf{x} : \mathbf{int}) \\ (\mathbf{t} = \mathbf{int}, \mathbf{x} : \mathbf{int}) \succeq [\mathbf{int}/\alpha]\,(\mathbf{t} = \alpha, \mathbf{x} : \alpha)\end{array}}{\vdash \mathrm{F}\ \mathrm{s} : \exists\emptyset.[\mathbf{int}/\alpha]\,(\mathbf{y} : \alpha).}$$

The semantic object of the application is obtained by a *second-order* substitution, substituting the *type* **int** for the quantified type variable α in the functor range. Conceptually, this derivation implicitly combines the following two steps. We first choose an appropriate instance of the polymorphic functor, obtaining a monomorphic function of type $(\mathbf{t} = \mathbf{int}, \mathbf{x} : \mathbf{int}) \rightarrow (\mathbf{y} : \mathbf{int})$. We then apply the monomorphic function to an argument in its domain, returning a result of type $(\mathbf{y} : \mathbf{int})$.

Finally, in MacQueen's interpretation a structure containing a substructure is modeled as a nested pair of terms. The type of a pair is a cross-product. Again, the need to type pairs using first-order existential quantification, i.e. dependent cross products, follows from the fact that the first component of a product may contain existentially quantified, and thus opaque, type components. For instance, consider wanting to express a function taking a pair, whose first component is a pair of a type and a term, and whose second component is a term of this type. Note that the first component corresponds to a substructure with a type component. We need to express the domain of our function as a cross product of a strong existential and some second type. Since the strong existential type does not reveal the identity of its type component, the only way to express this second type is by projection from the term inhabiting the strong existential. This means that the second component of a cross product must be allowed to depend on the term inhabiting its first component. Thus we need dependent cross products, i.e. first-order existential quantification over terms.

Example 9.2.3. Suppose we are trying to write a functor with the argument signature:

$$\begin{array}{rcl} \mathrm{S}' & \stackrel{\text{def}}{=} & \mathbf{sig\ structure\ Y} : \mathrm{S}; \mathbf{val\ y} : \mathbf{Y}.\mathbf{t\ end} \\ & \equiv & \mathbf{sig\ structure\ Y} : \mathbf{sig\ type\ t} : 0; \mathbf{val\ x} : \mathbf{t\ end}; \mathbf{val\ y} : \mathbf{Y}.\mathbf{t\ end}. \end{array}$$

Notice how the type of **y** is specified in terms of the type component of the substructure **Y**.

To be consistent with MacQueen's interpretation of signatures as types, this signature must correspond to the type of a nested pair, consisting of a term Y of type $[\![\mathrm{S}]\!]$, itself pairing a type with a term, together with another term of this type. Because the implementation of Y's type component is hidden in $[\![\mathrm{S}]\!]$, the only way to refer to it is by projection from the term Y, yielding the first-order existential type:

$$\begin{array}{rcl} [\![\mathrm{S}']\!] & \stackrel{\text{def}}{=} & \exists \mathrm{Y}{:}[\![\mathrm{S}]\!].\mathbf{Fst}\ \mathrm{Y} \\ & \equiv & \exists \mathrm{Y}{:}(\Sigma\alpha{:}0.\alpha).\mathbf{Fst}\ \mathrm{Y} \end{array}$$

Let's compare this type with the actual denotation of the signature S′:

$$\vdash \textbf{sig structure Y} : \mathrm{S}; \textbf{val y} : \textbf{Y.t end} \triangleright \Lambda\{\alpha\}.(\mathbf{Y} : (\mathbf{t} = \alpha, \mathbf{x} : \alpha), \mathbf{y} : \alpha).$$

Observe that the apparent first-order dependency of the type of **y** on the term **Y** has been eliminated in favour of a second-order dependency on the type parameter α: since α explicitly represents the denotation of **Y**'s type component, we can express the type of **y** *without* referring to **Y**.

In summary, by examining the actual semantics of Modules, we can see little evidence to support the claim that Standard ML's type structure is based on first-order dependent types.

9.2.2 Leroy's Modules and Harper and Lillibridge's Translucent Sums

In a sense, Leroy's module calculi [Ler94, Ler96b, Ler95], Harper and Lillibridge's translucent sums calculus [HL94] and its descendants [Lil97, SH96, HS97] are refinements of the analogy proposed by MacQueen. While abandoning MacQueen's use of the strong existential, they still interpret signatures as types. Moreover, because signatures can declare *opaque* type components, they also need to resort to the use of first-order dependent types. We will focus on Leroy's work, but the other systems are similar, and the comments in this section apply to all of them.

In MacQueen's approach, the type of a structure cannot reveal the denotations of its type components: the transparency of the structure's type components is achieved by inspection of the structure itself, violating the phase distinction. Like MacQueen, Leroy interprets signatures as the types of structures. However, by enriching the notion of signature to allow a mixture of opaque and manifest type declarations, Leroy can account for transparency, while preserving the phase distinction. Type components with opaque declarations are abstract. Type components with manifest declarations are transparent. In this way, the denotation of a structure's type component can be determined by inspection of the structure's signature, as long as the component has a manifest declaration in that signature.

Remark 9.2.2. Leroy use syntactic type phrases, not semantic objects, to classify term phrases. For instance, the classification judgement for module expressions has the form $\mathrm{C} \vdash \mathrm{m} : \mathrm{S}$, relating the module expression m to a syntactic signature, where C is a context mapping identifiers to *syntactic* type phrases. Because our syntax of type phrases is almost identical to Leroy's, we shall employ it when presenting examples in his semantics.

Example 9.2.4. Continuing with Example 9.2.1, in Leroy's calculus, the structure s is assigned the signature:

$$\vdash \mathrm{s} : \mathbf{sig\ type\ t} = \mathbf{int}; \mathbf{val\ x} : \mathbf{int\ end}.$$

Observe that **t** is declared to be manifestly equal to **int** in the type of s.

Leroy defines a context-dependent subtyping relation on signatures, written $\mathrm{C} \vdash \mathrm{S} \subseteq \mathrm{S}'$, that, in a sense, combines our separate notions of enrichment and realisation in a *single* relation.

Example 9.2.5. In Leroy's semantics, one can derive:

$$\vdash \mathbf{sig\ type\ t} = \mathbf{int}; \mathbf{val\ x} : \mathbf{int\ end} \quad \subseteq \quad \mathbf{sig\ val\ x} : \mathbf{int\ end},$$

reflecting the fact that, in our semantics, any semantic structure matching (the denotation of) the signature on the left also matches (the denotation of) the signature on the right by virtue of *enrichment*.

Furthermore, in Leroy's semantics, one can also derive:

$$\vdash \mathbf{sig\ type\ t} = \mathbf{int}; \mathbf{val\ x} : \mathbf{int\ end} \quad \subseteq \quad \mathbf{sig\ type\ t} : 0; \mathbf{val\ x} : \mathbf{t\ end},$$

reflecting the fact that, in our semantics, any semantic structure matching the signature on the left also matches the signature on the right by virtue of a *realisation*. Notice, however, that the realisation is left *implicit* in Leroy's subtyping judgement.

Using his subtyping relation, Leroy can interpret Standard ML's abstraction phrase $\mathrm{s} \setminus \mathrm{S}$ as a coercion to a supertype.

Example 9.2.6. For instance, combining the typing judgement:

$$\vdash \mathrm{s} : \mathbf{sig\ type\ t} = \mathbf{int}; \mathbf{val\ x} : \mathbf{int\ end},$$

with the subtyping judgement:

$$\vdash \mathbf{sig\ type\ t} = \mathbf{int}; \mathbf{val\ x} : \mathbf{int\ end} \quad \subseteq \quad \mathbf{sig\ type\ t} : 0; \mathbf{val\ x} : \mathbf{t\ end},$$

one can derive the signature of the abstraction:

$$\vdash \mathrm{s} \setminus \mathbf{sig\ type\ t} : 0; \mathbf{val\ x} : \mathbf{t\ end} : \mathbf{sig\ type\ t} : 0; \mathbf{val\ x} : \mathbf{t\ end}.$$

It's interesting to compare this with the semantic object of the abstraction in our semantics:

$$\vdash \mathrm{s} \setminus \mathbf{sig\ type\ t} : 0; \mathbf{val\ x} : \mathbf{t\ end} : \exists\{\alpha\}.(\mathbf{t} = \alpha, \mathbf{x} : \alpha).$$

In Leroy's semantics, the abstract type is represented by the opaque type component **t**, consequently the type of **x** depends on the component **t**. In our semantics, the abstract type has an *independent* representation as a quantified type variable, the denotation of **t** is apparent, and there is no dependency of **x**'s type on the identifier **t**. We will shortly see why the existence of an independent representation for the abstract type is significant.

Unfortunately, having combined the notions of enrichment and realisation into a *single* subtyping relation, Leroy can no longer account for the effect of merely curtailing a structure by a signature, forcing him to abandon the curtailment phrase. Recall that the semantics of abstraction and curtailment differ only in the treatment of the matching realisation. In an abstraction, the actual realisation is effectively forgotten and may be left implicit. In a curtailment, however, the actual realisation *is* required since it must be preserved. The problem with Leroy's definition of subtyping is that it leaves realisations *implicit*, accommodating abstractions, but ruling out a direct semantics for curtailment phrases.

Leroy needs dependent function spaces (i.e. functor signatures) to describe the types of functors. As in MacQueen's DL, the need for dependent function spaces arises from the use of signatures as types, coupled with the fact that signatures can contain opaque type components.

Example 9.2.7. In Leroy's system the body of the functor F from Example 9.2.2 has the signature:

$$\begin{aligned}\mathbf{X} : \mathbf{sig\ type\ t} : 0; \mathbf{val\ x} : \mathbf{t\ end} \vdash \mathbf{struct\ val\ y} = \mathbf{X.x\ end} :&\\ \mathbf{sig\ val\ y} : \mathbf{X.t\ end}.&\end{aligned}$$

Since the type component of **X** is declared opaquely in the context, the type occurrence **X.t** cannot be simplified any further. Hence the dependency of the body's type on the *term* **X** cannot be removed, and the functor F must be given a dependent type that is conveniently expressed using the dependent syntax of functor signatures[1]:

$$\vdash \mathrm{F} : \mathbf{funsig}(\mathbf{X}{:}\mathbf{sig\ type\ t} : 0; \mathbf{val\ x} : \mathbf{t\ end})\mathbf{sig\ val\ y} : \mathbf{X.t\ end}.$$

Although functors have dependent types, Leroy must avoid performing first-order substitution when typechecking functor applications; this is because the syntax of types is not closed under the substitution of module expressions for module identifiers. He does this by adopting a novel

[1]Note that, although their syntax is the same, functor signatures play a different role in Leroy's semantics: in Leroy's semantics, a functor signature is a type classifying terms; in our system, a functor signature denotes a parameterised type.

elimination rule: a functor may only be applied if it can first be given a non-dependent supertype using a covariant subtyping rule.

Example 9.2.8. In Leroy's system, to type the application F s we first need to determine a supertype for F that is both non-dependent and has the type of s as its domain. To this end, we can show that the original, dependent functor signature:

$$\mathbf{funsig}(\mathbf{X}{:}\mathbf{sig}\ \mathbf{type}\ \mathbf{t} : 0; \mathbf{val}\ \mathbf{x} : \mathbf{t}\ \mathbf{end})\mathbf{sig}\ \mathbf{val}\ \mathbf{y} : \mathbf{X}.\mathbf{t}\ \mathbf{end}$$

is a subtype of the non-dependent supertype:

$$\mathbf{sig}\ \mathbf{type}\ \mathbf{t} = \mathbf{int}; \mathbf{val}\ \mathbf{x} : \mathbf{int}\ \mathbf{end} \to \mathbf{sig}\ \mathbf{val}\ \mathbf{y} : \mathbf{int}\ \mathbf{end}$$

by the following reasoning. First, observe that the domain of the supertype is a subtype of the original domain:

$$\vdash \mathbf{sig}\ \mathbf{type}\ \mathbf{t} = \mathbf{int}; \mathbf{val}\ \mathbf{x} : \mathbf{int}\ \mathbf{end} \subseteq \mathbf{sig}\ \mathbf{type}\ \mathbf{t} : 0; \mathbf{val}\ \mathbf{x} : \mathbf{t}\ \mathbf{end}.$$

Second, under the more informative assumption that **X** belongs to this subdomain, we can show that the original range is a subtype of the supertype's range:

$$\mathbf{X} : \mathbf{sig}\ \mathbf{type}\ \mathbf{t} = \mathbf{int}; \mathbf{val}\ \mathbf{x} : \mathbf{int}\ \mathbf{end} \vdash \mathbf{sig}\ \mathbf{val}\ \mathbf{y} : \mathbf{X}.\mathbf{t}\ \mathbf{end} \subseteq \\ \mathbf{sig}\ \mathbf{val}\ \mathbf{y} : \mathbf{int}\ \mathbf{end}.$$

Note that to establish this last judgement we need to exploit the fact that, in the subdomain, **X**'s type component is manifestly equal to **int**. Finally, combining these two facts, the subtyping rule for functor signatures derives:

$$\vdash \quad \mathbf{funsig}(\mathbf{X}{:}\mathbf{sig}\ \mathbf{type}\ \mathbf{t} : 0; \mathbf{val}\ \mathbf{x} : \mathbf{t}\ \mathbf{end})\mathbf{sig}\ \mathbf{val}\ \mathbf{y} : \mathbf{X}.\mathbf{t}\ \mathbf{end} \quad \subseteq \\ \mathbf{sig}\ \mathbf{type}\ \mathbf{t} = \mathbf{int}; \mathbf{val}\ \mathbf{x} : \mathbf{int}\ \mathbf{end} \to \mathbf{sig}\ \mathbf{val}\ \mathbf{y} : \mathbf{int}\ \mathbf{end}.$$

By the subsumption rule, we can now derive that F also has the non-dependent type:

$$\vdash \mathrm{F} : \mathbf{sig}\ \mathbf{type}\ \mathbf{t} = \mathbf{int}; \mathbf{val}\ \mathbf{x} : \mathbf{int}\ \mathbf{end} \to \mathbf{sig}\ \mathbf{val}\ \mathbf{y} : \mathbf{int}\ \mathbf{end}.$$

Clearly, the structure s is in F's domain, since:

$$\vdash \mathrm{s} : \mathbf{sig}\ \mathbf{type}\ \mathbf{t} = \mathbf{int}; \mathbf{val}\ \mathbf{x} : \mathbf{int}\ \mathbf{end}.$$

Moreover, because the range of F no longer depends on the element of F's domain, the application F s can be assigned the result signature:

$$\vdash \mathrm{F}\ \mathrm{s} : \mathbf{sig}\ \mathbf{val}\ \mathbf{y} : \mathbf{int}\ \mathbf{end},$$

using the standard elimination rule for *non-dependent* functions, i.e. *without* having to substitute the term s into F's range.

Unfortunately, using the subtyping trick to eliminate dependencies doesn't always work well, and sometimes doesn't work at all. The success of the trick relies crucially on the pre-condition that every abstract type, declared in the domain of the functor and propagated to its range, is declared manifestly in the signature of the actual argument.

Example 9.2.9. Here's an example in which the trick works, but with a counter-intuitive result. Consider the application F (s \ S). Recall that its argument has an opaque signature:

$$\vdash (\mathrm{s} \setminus \mathrm{S}) : \mathbf{sig}\ \mathbf{type}\ \mathbf{t} : 0; \mathbf{val}\ \mathbf{x} : \mathbf{t}\ \mathbf{end}.$$

Now the only applicable, non-dependent supertype that we can find for the functor is:

$$\vdash \mathrm{F} : \mathbf{sig}\ \mathbf{type}\ \mathbf{t} : 0; \mathbf{val}\ \mathbf{x} : \mathbf{t}\ \mathbf{end} \rightarrow \mathbf{sig}\ \ \mathbf{end}.$$

Applying F with this type yields:

$$\vdash \mathrm{F}\ (\mathrm{s} \setminus \mathrm{S}) : \mathbf{sig}\ \ \mathbf{end}.$$

Observe that the value component **y** has disappeared from the result. Now compare this signature with the semantic object assigned to the term:

$$\vdash \mathrm{F}\ (\mathrm{s} \setminus \mathrm{S}) : \exists\alpha.(\mathbf{y} : \alpha)$$

In our semantics, the value component is still available. Of course, in this example, the value component is useless because there are no operations that can manipulate this value. However, it is easy to construct larger examples where the result is a full-blown abstract data type together with useful operations.

We can explain this difference in behaviour as follows. In Leroy's calculus, the only way to represent an abstract type is as a reference to an opaque type component of a structure, i.e. either as a type identifier, or as the projection of a type identifier from a *path* of structure identifiers declared in the context. The problem with the functor application above is that it propagates an abstract type from the context of the functor body, in which it has a syntactic representation, to the context of the functor application, in which it does not. Consequently, any reference to the anonymous abstract type in the result type must be removed. In this case, the reference is removed by forgetting the **y**-component of the result.

In our semantics, an abstract type, whether denoted by a type component or not, retains an independent semantic representation as an existentially quantified type variable. Indeed, examining the semantic object of the application we can see that the existentially quantified variable does not appear as the denotation of a type component. However, we can still use it to describe the type of the value component **y**. Of course, for the reasons we discussed in Chapter 6, the type of the application fails to have a syntactic description in our semantics, causing problems with separate compilation. In Leroy's system, the type of the application does have a complete syntactic description, but the type is less informative.

Example 9.2.10. Here is an example where the trick doesn't work at all. Let's define the curried functor:

$$\begin{array}{rcl} \mathrm{G} & \equiv & \mathbf{functor}(\mathbf{X} : \mathbf{sig\ type\ t} : 0; \mathbf{val\ x} : \mathbf{t\ end}) \\ & & \quad\quad \mathbf{functor}(\mathbf{Y} : \mathbf{sig\ val\ y} : \mathbf{X}.\mathbf{t\ end})\mathbf{struct\ \ end}, \end{array}$$

with type:

$$\begin{array}{rcl} \vdash \mathrm{G} & : & \mathbf{funsig}(\mathbf{X}{:}\mathbf{sig\ type\ t} : 0; \mathbf{val\ x} : \mathbf{t\ end}) \\ & & \quad\quad \mathbf{funsig}(\mathbf{Y}{:}\mathbf{sig\ val\ y} : \mathbf{X}.\mathbf{t\ end})\mathbf{sig\ \ end}. \end{array}$$

The problem arises when we try to type the partial application G $(\mathrm{s} \setminus \mathrm{S})$. In the previous example, the application of F to $(\mathrm{s} \setminus \mathrm{S})$, we could remove the dependency of F's range on its domain by forgetting the **y**-component evincing the dependency. We could do this because the declaration of **y** occurs in a covariant position in F's range. However, in the application of G, the dependency cannot be removed because the declaration of **y** occurs in a *contravariant* position in G's range. In Leroy's semantics, the application is rejected as ill-typed.

We would argue that this behaviour is counter-intuitive: the application cannot be typed, *even though* the signature of the actual argument is a subtype of the functor's domain. Indeed, in our semantics, the application does have a type:

$$\vdash \mathrm{G}\ (\mathrm{s} \setminus \mathrm{S}) : \exists\{\alpha\}.\forall\emptyset.(\mathbf{y} : \alpha) \to ().$$

Again, notice how the existentially quantified type variable is used to represent an abstract type that, being anonymous, cannot be accounted for in Leroy's system.

Even if we accept the behaviour in the previous examples, there is a more serious problem with Leroy's approach. In some situations, a functor may have *too many* non-dependent supertypes, without a principled way to choose between them. Leroy's semantics lacks the *principal typing property*. From a practical perspective, this means that a programmer cannot always rely on the type-checker determining a type commensurate with her expectations.

Example 9.2.11. This example illustrates the absence of principal types. Let's define the higher-order functor:

$$\begin{aligned} \mathrm{H} \quad \equiv \quad & \mathbf{functor}(\mathbf{X} : \mathbf{sig\ type\ t} : 0; \mathbf{val\ x} : \mathbf{t\ end}) \\ & \quad \mathbf{functor}(\mathbf{Y} : \mathbf{sig\ type\ u} : 1\ \mathbf{end})\mathbf{struct\ type\ v} = \mathbf{Y}.\mathbf{u}(\mathbf{X}.\mathbf{t})\ \mathbf{end}, \end{aligned}$$

which has type:

$$\begin{aligned} \vdash \mathrm{H} \quad : \quad & \mathbf{funsig}(\mathbf{X}{:}\mathbf{sig\ type\ t} : 0; \mathbf{val\ x} : \mathbf{t\ end}) \\ & \qquad \mathbf{funsig}(\mathbf{Y}{:}\mathbf{sig\ type\ u} : 1\ \mathbf{end})\mathbf{sig\ type\ v} = \mathbf{Y}.\mathbf{u}(\mathbf{X}.\mathbf{t})\ \mathbf{end} \end{aligned}$$

Now consider the partial application of H to the abstraction (s \ S). As in the previous examples, to type this application we need to remove the dependency of H's range on the identifier **X**. In other words, we need to find a supertype of

$$\mathbf{funsig}(\mathbf{Y}{:}\mathbf{sig\ type\ u} : 1\ \mathbf{end})\mathbf{sig\ type\ v} = \mathbf{Y}.\mathbf{u}(\mathbf{X}.\mathbf{t})\ \mathbf{end}, \qquad (\star)$$

that doesn't mention **X**. The contra-variant subtyping rule gives us *two* ways in which to proceed.

The obvious solution is to replace the *range* of the functor signature by a supertype not depending on **X**:

$$\begin{aligned} & \mathbf{funsig}(\mathbf{Y}{:}\mathbf{sig\ type\ u} : 1\ \mathbf{end})\mathbf{sig\ type\ v} = \mathbf{Y}.\mathbf{u}(\mathbf{X}.\mathbf{t})\ \mathbf{end} \\ \subseteq\ & \mathbf{funsig}(\mathbf{Y}{:}\mathbf{sig\ type\ u} : 1\ \mathbf{end})\mathbf{sig\ type\ v} : 0\ \mathbf{end}. \end{aligned}$$

The less obvious solution is to replace the *domain* of the functor signature by a subtype that enables us to eliminate the dependency in the range. In particular, if we choose some definable type Λ'a.u, such that neither **X** nor $'$a occur in u, then we can replace the opaque declaration of **u** by a manifest declaration equating **u** with Λ'a.u. Exploiting this equation then allows us to remove the dependency in the range:

$$\begin{aligned} & \mathbf{funsig}(\mathbf{Y}{:}\mathbf{sig\ type\ u} : 1\ \mathbf{end})\mathbf{sig\ type\ v} = \mathbf{Y}.\mathbf{u}(\mathbf{X}.\mathbf{t})\ \mathbf{end} \\ \subseteq\ & \mathbf{funsig}(\mathbf{Y}{:}\mathbf{sig\ type\ u} = \Lambda'\mathrm{a.u}\ \mathbf{end})\mathbf{sig\ type\ v} = \mathbf{Y}.\mathbf{u}(\mathbf{X}.\mathbf{t})\ \mathbf{end} \\ \equiv\ & \mathbf{funsig}(\mathbf{Y}{:}\mathbf{sig\ type\ u} = \Lambda'\mathrm{a.u}\ \mathbf{end})\mathbf{sig\ type\ v} = \mathrm{u}\ \mathbf{end}. \end{aligned}$$

```
module F = F;

module X = struct type t = int; val x = 1 end;
module Y = X;
module Z = struct type t = int; val x = 2 end;
module U = struct type t = bool; val x = true end;

module A1 = F X;
module A2 = F X;
module B = F Y;
module C = F Z;
module D = F U;
```

Figure 9.1: An example illustrating the difference between our notion of applicative functors and Leroy's.

The problem is that these solutions are totally unrelated: neither is a subtype of the other, nor is there a *non-dependent* supertype of ($\star$) that is a subtype of both. So which do we choose?

In our semantics, the application is given the unique and thus principal type:

$$\vdash \mathrm{H}\ (\mathrm{s} \setminus \mathrm{S}) : \exists\{\alpha\}.\forall\{\beta\}.(\mathbf{u} = \beta) \rightarrow (\mathbf{v} = (\beta\ \alpha)).$$

Notice how the anonymous abstract type is represented by the type variable α. In Leroy's system, the anonymous type can't be represented and must be eliminated. Recast in our system, the first solution we gave corresponds to preserving the functor's polymorphism but narrowing its range; while the second solution corresponds to restricting the functor's polymorphism but preserving its range.

9.2.3 Leroy's applicative functors

In Chapter 5 we gave a semantics for applicative functors. The terminology is borrowed from Leroy, who has proposed an applicative version of his own module calculus [Ler95]. Although similar in spirit, the two notions of applicative functor are subtly different.

Roughly speaking, in our semantics, functors are applicative in the sense that two different applications of the same functor at the same *realisation* yield equivalent abstract types. In Leroy's semantics, two different applica-

$$\begin{array}{l}
\mathbf{module}\ \lfloor_{\forall\{\beta\}.(\mathbf{t}=\beta,\mathbf{x}:\beta)\to(\mathbf{t}=(\alpha\ \beta),\mathbf{x}:(\alpha\ \beta))}\mathbf{F} = \mathrm{F};\\[1em]
\mathbf{module}\ \lfloor_{(\mathbf{t}=\mathbf{int},\mathbf{x}:\mathbf{int})}\mathbf{X} = \mathbf{struct\ type\ t} = \mathbf{int};\mathbf{val}\ \mathrm{x} = 1\ \mathbf{end};\\
\mathbf{module}\ \lfloor_{(\mathbf{t}=\mathbf{int},\mathbf{x}:\mathbf{int})}\mathbf{Y} = \mathbf{X};\\
\mathbf{module}\ \lfloor_{(\mathbf{t}=\mathbf{int},\mathbf{x}:\mathbf{int})}\mathbf{Z} = \mathbf{struct\ type\ t} = \mathbf{int};\mathbf{val}\ \mathrm{x} = 2\ \mathbf{end};\\
\mathbf{module}\ \lfloor_{(\mathbf{t}=\mathbf{bool},\mathbf{x}:\mathbf{bool})}\mathbf{U} = \mathbf{struct\ type\ t} = \mathbf{bool};\mathbf{val}\ \mathrm{x} = \mathbf{true}\ \mathbf{end};\\[1em]
\mathbf{module}\ \lfloor_{(\mathbf{t}=(\alpha\ \mathbf{int}),\mathbf{x}:(\alpha\ \mathbf{int}))}\mathbf{A1} = \mathbf{F}\ \mathbf{X};\\
\mathbf{module}\ \lfloor_{(\mathbf{t}=(\alpha\ \mathbf{int}),\mathbf{x}:(\alpha\ \mathbf{int}))}\mathbf{A2} = \mathbf{F}\ \mathbf{X};\\
\mathbf{module}\ \lfloor_{(\mathbf{t}=(\alpha\ \mathbf{int}),\mathbf{x}:(\alpha\ \mathbf{int}))}\mathbf{B} = \mathbf{F}\ \mathbf{Y};\\
\mathbf{module}\ \lfloor_{(\mathbf{t}=(\alpha\ \mathbf{int}),\mathbf{x}:(\alpha\ \mathbf{int}))}\mathbf{C} = \mathbf{F}\ \mathbf{Z};\\
\mathbf{module}\ \lfloor_{(\mathbf{t}=(\alpha\ \mathbf{bool}),\mathbf{x}:(\alpha\ \mathbf{bool}))}\mathbf{D} = \mathbf{F}\ \mathbf{U};
\end{array}$$

Figure 9.2: The example of Figure 9.1, annotated with semantic objects according to our applicative semantics of Chapter 5.

```
module ⌊F = F;
        funsig(X:sig type t:0;val x:t end)sig type t:0;val x:t end

module ⌊X = struct type t = int; val x = 1 end;
        sig type t = int;val x:int end
module ⌊Y = X;
        sig type t = int;val x:int end
module ⌊Z = struct type t = int; val x = 2 end;
        sig type t = int;val x:int end
module ⌊U = struct type t = bool; val x = true end;
        sig type t = bool;val x:bool end

module ⌊A1 = F X;
        sig type t = (F X).t;val x:t end
module ⌊A2 = F X;
        sig type t = (F X).t;val x:t end
module ⌊B = F Y;
        sig type t = (F Y).t;val x:t end
module ⌊C = F Z;
        sig type t = (F Z).t;val x:t end
module ⌊D = F U;
        sig type t = (F U).t;val x:t end
```

Figure 9.3: The example of Figure 9.1, annotated with types according to the applicative semantics of Leroy.

tions of the same functor path to the same *argument path* yield equivalent abstract types.

The consequences of this difference in semantics are best illustrated by example. Let's define the following functor:

$$\begin{array}{lcl} \mathrm{F} & \overset{\mathrm{def}}{=} & \mathbf{functor}(\mathbf{X} : \mathbf{sig\ type\ t} : 0; \mathbf{val\ x} : \mathbf{t\ end}) \\ & & \qquad\qquad \mathbf{X} \setminus \mathbf{sig\ type\ t} : 0; \mathbf{val\ x} : \mathbf{t\ end}. \end{array}$$

Observe that F introduces an abstract type in its body.

In our applicative semantics, F has the following semantic object:

$$\vdash \mathrm{F} : \exists\{\alpha\}.\forall\{\beta\}.(\mathbf{t} = \beta, \mathbf{x} : \beta) \to (\mathbf{t} = (\alpha\ \beta), \mathbf{x} : (\alpha\ \beta)),$$

where α represents the abstract argument-result type dependency of F. Intuitively, α represents this dependency as a function of the argument's *type component*. If we bind the module expression F to a module identifier **F**, then the existential quantifier is eliminated, and the type of **F** is recorded in the context as the assumption:

$$\ldots, \mathbf{F} : \forall\{\beta\}.(\mathbf{t} = \beta, \mathbf{x} : \beta) \to (\mathbf{t} = (\alpha\ \beta), \mathbf{x} : (\alpha\ \beta)), \ldots$$

for a fixed, but abstract, dependency α.

In Leroy's applicative semantics, the type of F is:

$$\vdash \mathrm{F} : \mathbf{funsig}(\mathbf{X}{:}\mathbf{sig\ type\ t} : 0; \mathbf{val\ x} : \mathbf{t\ end})\mathbf{sig\ type\ t} : 0; \mathbf{val\ x} : \mathbf{t\ end}.$$

If we bind this module expression to a module identifier **F**, then its type is simply recorded in the context as the assumption:

$$\ldots, \mathbf{F} : \begin{array}{l} \mathbf{funsig}(\mathbf{X}{:}\mathbf{sig\ type\ t} : 0; \mathbf{val\ x} : \mathbf{t\ end}) \\ \qquad\qquad \mathbf{sig\ type\ t} : 0; \mathbf{val\ x} : \mathbf{t\ end} \end{array}, \ldots$$

However, each *occurrence* of the module identifier **F** will be given the same *strengthened* type:

$$\begin{array}{l} \ldots \vdash \mathbf{F} : \mathbf{funsig}(\mathbf{X}{:}\mathbf{sig\ type\ t} : 0; \mathbf{val\ x} : \mathbf{t\ end}) \\ \qquad\qquad\qquad \mathbf{sig\ type\ t} = (\mathbf{F\ X}).\mathbf{t}; \mathbf{val\ x} : \mathbf{t\ end}. \end{array}$$

Intuitively, the projection $(\mathbf{F\ X}).\mathbf{t}$ denotes the argument-result type dependency of **F** as a function of its entire *argument*, **X**, not merely as a function of **X**'s type component. Note that Leroy's strengthening operation corresponds to a form of existential elimination.

In Figure 9.1, we define such a functor **F**, some sample argument structures **X**, **Y**, **Z** and **U**, and the structures **A1**, **A2**, **B**, **C** and **D** resulting from the application of **F** to these arguments. The difference between our notion of applicative functor and Leroy's is illustrated by the equalities that hold between the type components of **A1**, **A2**, **B**, **C** and **D**.

In our semantics, the set of type components:

$$\{\mathbf{A1.t}, \mathbf{A2.t}, \mathbf{B.t}, \mathbf{C.t}, \mathbf{D.t}\}$$

is partitioned into *two* disjoint equivalence classes:

$$\{\mathbf{A1.t}, \mathbf{A2.t}, \mathbf{B.t}, \mathbf{C.t}\} \quad \{\mathbf{D.t}\}$$

corresponding, respectively, to the representative denotations (α **int**) and (α **bool**).

Figure 9.2 clearly shows why this is the case. In the definitions of **A1**, **A2**, **B**, and **C**, the parameter β of **F** is realised by the same type **int**, because each actual argument of **F** implements the type component **t** as **int**. Consequently, in each definition, the resulting type component receives the same denotation (α **int**). In the definition of **D**, on the other hand, the functor must be applied with a different realisation (β is realised by **bool**), and the resulting type component receives the distinct denotation (α **bool**).

In Leroy's semantics, the set is partitioned into *four* disjoint equivalence classes:

$$\{\mathbf{A1.t}, \mathbf{A2.t}\} \quad \{\mathbf{B.t}\} \quad \{\mathbf{C.t}\} \quad \{\mathbf{D.t}\}$$

corresponding, respectively, to the representative type projections (**F X**).**t**, (**F Y**).**t**, (**F Z**).**t**, and (**F U**).**t**.

Figure 9.3 shows why this the case, by revealing the types assigned to each module identifier in Leroy's semantics. The type components of **A1** and **A2** are equivalent because, in both definitions, the functor **F** is applied to the same *path* **X** (in a generative semantics, the type components of **A1** and **A2** would be distinct). The remaining type components are all distinct, because in every case, the functor is applied to a *different* path.

This raises the question of which behaviour is preferable. One argument in favour of Leroy's semantics, and against ours, is that the type **C.t** should be distinct from **A1.t** (as well as **A2.t** and **B.t**) because the arguments of the applications **F X** and **F Z** differ on their implementation of the *value* component **x**. This behaviour can be important if the interpretation of the abstract type returned by **F** depends on the value of its argument's **x**-component. For instance, if **F** returned an abstract data type of finite sets, where each set is represented uniquely as a sorted list of elements, and **x** is the comparison function used to sort the elements, then it is desirable that different choices of **x** result in distinct abstract types. An opposing argument, against Leroy's semantics and in favour of ours, is that applying a functor to a structure on one occasion, and a renaming of that structure on another, shouldn't affect the compatibility of the resulting types. Note that **Y** is merely a renaming of **X**, but the abstract types returned by **F X** and **F Y** are distinct in Leroy's semantics; they are equivalent in ours.

9.3 Further Research

We close by considering some directions for further research.

In order to simplify the presentation of Modules, we have chosen to omit Standard ML's facility for defining signature identifiers abbreviating (the denotations) of signature expressions. For our theoretical study of Modules, this omission is insignificant, since an occurrence of a signature identifier can always be replaced by an in-line expansion of its definition. However, for programming convenience, signature abbreviations should be supported, since they drastically reduce syntactic clutter. With the introduction of signature definitions, it also makes sense to consider adding syntactic support for qualifying a signature expression's denotation by further instantiation and coalescing of its type parameters. Signature abbreviations and a form of signature qualification are supported in our implementation [Rus98a]. Both can be described as simple extensions of the syntax and static semantics presented in this thesis.

One obvious omission of this thesis is the definition of a dynamic semantics for Modules. It is straightforward to define an untyped, call-by-value semantics in the style of Standard ML [MTH90, MTH96, MG93]. A dynamic semantics would enable us to state and attempt to prove a type soundness theorem for the language: that well-typed programs do not "go wrong". Most importantly, this work would complete the semantic justification for First-Class Modules that we sketched in Section 7.4.1.

We need to develop a proof of correctness for the type inference and unification algorithms of Chapter 8. In the absence of a proof, what justification do we have for believing in their correctness? Strictly speaking, none. However, we have striven to obtain a declarative description of the algorithms that clearly reveals their roots in the verified algorithms for ML type inference and the verified algorithms of Chapter 5. This should at least increase our confidence in the algorithms. What's more, empirical evidence gained with our prototype implementation supports the conjecture that the algorithms are correct, but also suggests that a more efficient implementation is required. One benefit of our declarative description is that it should make the verification task easier, at least in principle. In practice, the major obstacle for a human prover is the number of cases to be considered; for a mechanical prover, the number of details to be formalised. Finally, we should point out that we expect the termination proof for the unification algorithm to be more difficult in the presence of first-class modules: the need to perform matching on value types causes the introduction of new unification variables *during* unification, foiling the measure traditionally used to establish termination of ordinary first-order unification.

From the perspective of Type Theory, it would be nice to give a more conclusive demonstration that Modules can be understood without resorting to dependent types. We strongly believe that it is possible to use the ideas in this thesis to define a type-directed translation from Modules (especially with Core-ML as a core language) into a standard type theory based on the simply-typed λ-calculus with higher-order parameterised types, universal and existential quantification over types, and records. Since the standard type theories combining all of these features are explicitly typed, the main difficulty will lie in defining a translation that makes the implicit manipulation of quantifiers and modules subtyping explicit, in a coherent manner. The problem is exacerbated by the applicative semantics of higher-order functors, since the implicit skolemisation of existential types requires the equivalent of an axiom of choice.

One distinguishing aspect of our work is that it maintains the distinction between syntactic type phrases and semantic objects. Although this can be criticised as ugly, it also offers an advantage: our framework is readily compatible with the extension to Standard ML's original notion of *structure generativity* and *structure sharing*. Although this feature has been removed in the revised semantics of Standard ML, it is still of interest, since sharing of structures is a stronger property than mere sharing of types: it provides a static guarantee of the identity of values. Indeed, the existing semantics of structure generativity is closely related to the semantics of type generativity. We believe that our explanation of type generativity can be adapted to give a treatment of structure generativity as existential quantification over static structure names. Moreover, the idea of adopting an applicative semantics of type generativity to support higher-order functors should carry over to structure generativity as well, promising a treatment of structure generativity in the higher-order case. Traditionally, the major difficulty presented by the semantics of structure sharing lies in determining the principal denotation of a signature that is qualified by structure sharing constraints. However, this problem is very similar to the one encountered in determining the principal denotation of a signature that is qualified by type sharing constraints. In the latter case, it is known that the problem of ensuring principality becomes trivial if type sharing constraints are abandoned in favour of definitional type specifications in signatures. It is very plausible that structure sharing constraints may also be replaced by a simpler form of specification that eliminates the principality problem.

Finally, in order to make our results widely available we need to transfer them to both the full definition of Standard ML and to an existing implementation. Unfortunately, Standard ML's Core is a much richer language than Core-ML. Moreover, the actual semantics of Standard ML does not separate the treatment of Modules and Core as rigorously as we do. The main difficulty in transferring our results to Standard ML's semantics lies in separating the essential from the non-essential interactions between Standard ML's Core and Modules. Non-essential interactions are those that succumb to alternative treatments directly in the semantics of the Core. The facility for defining mutually recursive types and mutually recursive functions fall in this class, since these can be accounted for by other means using n-ary fixed-point operators. Other interactions may require proper extensions to the Modules language presented here. The distinction between definable types admitting equality and ordinary definable types may be one such feature. While the distinction itself can be encoded in the notion of Core *kind*, Standard ML also permits a definable type admitting equality to be regarded as one that does not. Accommodating this form of subsumption requires a notion of *subkinding* on definable types, which we have not addressed.

Chapter 10

Epilogue

After I completed my thesis [Rus98b], Don Sannella gave me the opportunity to adapt my extensions to full Standard ML (SML) and implement them in an existing compiler. I chose Sergei Romanenko and Peter Sestoft's excellent Moscow ML[RRS00], mainly for its popularity, portability and simple architecture. Moscow ML is a bytecode compiler that offers a separate compilation model similar to the one discussed in Chapter 6. My starting point, Moscow ML V1.44, only supported the Core language of SML. It used flat structures as compilation units, with cut-down signatures describing their interfaces, but the other features of Modules such as nested structures and SML's functors were not yet implemented. I spent my first two months editing the sources of the Definition of Standard ML [MTH96], adapting my extensions to the full language. At the same time, I took the opportunity to rationalise the existing definition. With a preliminary design document in hand, it took me another ten months to implement the extended Modules language on top of Moscow ML. Moscow ML compiles to an untyped intermediate language based on an extended lambda-calculus. This could easily be used to implement Modules as well as Core language constructs, and meant that most of my coding effort was concentrated on the front-end of the compiler, i.e. parsing and type-checking. I never intended to develop more than a prototype, but Peter Sestoft kindly invited me to Denmark to discuss my modifications. Much to my surprise, Peter adopted the changes; they were integrated into the official sources and released to an unsuspecting public as Moscow ML Version 2.00. This collaborative work with Peter and Ken Friis-Larsen, another co-developer of Moscow ML, has been by far the most rewarding aspect of my thesis work.

Although rooted in the proposals of this thesis, the SML extensions available in Moscow ML differ slightly in design and generality. The purpose of this epilogue is to informally motivate and document these differences, relating them to Mini-SML where possible. A formal definition of the extensions that I used as my design document is available on request [Rus00c]. The publicly available Moscow ML documentation [RRS00] defines the precise syntax of the extensions; this syntax is used here for any code examples.

10.1 Moscow ML's Extensions to Standard ML

In brief, the Moscow ML Modules language extends the SML Modules language with the following features:

- higher-order functors: a functor may be defined within a structure, passed as an argument to another functor, or returned as the result of a functor (Section 10.1.1);
- applicative as well as standard generative functors (Section 10.1.2);
- transparent and opaque functor signatures (Sections 10.1.3 and 10.1.4);
- generalised projections of types from arbitrary, statically checked but unevaluated, module expressions (see Section 10.1.5);
- first-class modules: structures and functors may be packed and then handled as Core language values, which may then be unpacked as structures or functors again using a more natural and expressive elimination construct than the one described in Chapter 7 (Section 10.1.6);
- recursive modules: structures and signatures may be recursively defined supporting cross-module recursion of both (data)types and values (Section 10.1.7);
- minor relaxations of miscellaneous SML restrictions; (Section 10.1.8).

My primary design goal of the Moscow ML Modules language was that it should be a conservative extension of SML: all programs that are accepted by SML are accepted by Moscow ML, with the same static and dynamic semantics. Because I have relaxed unnecessary restrictions in the SML semantics, the converse is not true: Moscow ML accepts some syntactically valid SML programs that would be rejected under the SML static semantics, despite being perfectly type safe. The design goal has had a number of minor consequences on the syntax of the extensions, which deviates from the presentation in earlier chapters to avoid introducing new keywords and grammatical ambiguities. More significantly, it has forced me to generalise the semantics of higher-order functors described in Chapter 5 to accommodate standard generative functors as well as non-standard applicative ones. I have also taken the opportunity to revise the semantics of first-class modules slightly, making their use more convenient for programmers, especially those working in interactive sessions [Rus00a, Rus00b]. Recursive modules are an entirely new extension, built on the foundations of this thesis, but introduced only briefly here; see [Rus01] for a complete formalisation in the spirit of Mini-SML.

10.1.1 Higher-Order Modules

In Chapter 5, when moving from first-order to higher-order functors, we took the liberty of making functors applicative rather than generative, arguing that applicative functors provide better support for higher-order modules programming. Be that as it may, adopting this change in Standard ML would occasionally, and confusingly, contravene an existing programmer's expectations, by identifying more types than the generative semantics. In Moscow ML, instead of switching to an exclusively applicative semantics, I decided to support both the existing generative semantics, with the same syntax, as well as the novel applicative semantics, using new syntax.

In SML Modules, structures and functor bodies cannot declare functors. In Moscow ML Modules, they can:

```
functor F1(S : sig type t val x: t end) =
   struct functor G(T : sig type u  val y: u end) =
             struct val pair = (S.x, T.y) end
   end
structure R11 = F1(struct type t=int val x=177 end)
structure R12 = R11.G(struct type u=string val y="abc" end)
val (a, b) = R12.pair
```

For compatibility with SML, Moscow ML retains the distinction between functor and structure bindings, instead of using the more uniform **module**-bindings of higher-order Mini-SML. Moreover, in keeping with SML, structure and functor identifiers continue to reside in separate namespaces.

A functor that returns a functor can, of course, be *curried*, avoiding the creation of an intermediate structure:

```
functor F2(S:sig type t  val x : t end)
```

```
             (T:sig type u  val y : u end) =
                struct val pair = (S.x, T.y) end
structure R2 =
   F2(struct type t=int val x=177 end)
        (struct type u=string val y="abc" end)
val (a, b) = R2.pair
```

The definition of F2 above is syntactic sugar for binding F2 to an *anonymous* functor:

```
functor F2 =
 functor(S : sig type t  val x : t end) =>
    functor(T : sig type u  val y : u end) =>
       struct val pair = (S.x, T.y) end
```

A functor may be declared to take another functor as an argument, whose type is specified using a functor signature:

```
(* G is a functor signature *)
signature G = functor(X:sig type t=int val x: t end)->
                 sig type u val y : u end

(* F3 takes the functor F as an argument, and applies it *)
functor F3(F:G) = F(struct type t=int val x=177 end)

(* R3 is the result of F3 applied to an anonymous functor *)
structure R3 =
   F3(functor(X:sig type t=int val x:t end)=>
         struct type u = X.t * X.t val y= (X.x,X.x) end)
```

As expected, wherever a functor of a certain type is required, one can instead supply a functor that has more a general type, that is, one which is more polymorphic, expects a less general argument, or produces a more general result:

```
(* F3 is applied to a more general functor than it requires *)
structure R4 =
   F3(functor(X:sig type t end)=>
         struct type u = X.t val y = 1 val z = [] end)
```

SML's separation of the name spaces for functors and structures means that it is perfectly legal to re-use the same name for both a structure and a functor, without one hiding the other:

```
structure M = struct end
functor M() = struct end
structure N1 = M    (* structure N1 bound to structure M  *)
functor   N2 = M    (* functor N1 bound to functor M  *)
structure N3 = M(M) (* functor M applied to structure M *)
```

However, in Moscow ML, when another functor, say `P`, simply returns the identifier `M`, as in:

```
functor P () = M
```

then it is not clear whether `M` refers to the structure `M` or the functor `M`. In this ambiguous case, Moscow ML always interprets `M`, on its own, as a structure, but one can explicitly write `op M` to refer to the functor `M` instead:

```
functor P () = M       (* P returns the structure M *)
functor Q () = op M    (* Q returns the functor M *)
```

The keyword **op** has no effect in unambiguous contexts, namely bindings, functor applications and the bodies of explicit signature constraints, where the expected module type of the identifier resolves the syntactic ambiguity.

10.1.2 Generative and Applicative Functors

In SML Modules, all functors are generative. If the body of a generative functor `FG` declares a datatype or opaque type `t`, then two applications of `FG` will create two structures `SG1` and `SG2` with distinct types `SG1.t` and `SG2.t`:

```
(* FG is generative *)
functor FG (S : sig end) = struct datatype t = C end
structure SG1 = FG() and SG2 = FG()
val res = if true then SG1.C else SG2.C (* ill-typed *)
```

Recall that a conditional expression requires both branches to have equivalent types, so the last declaration above is well-typed only if the type `SG1.t` is equivalent to `SG2.t`.

If functors had an *applicative*, not generative, semantics, the two types would be equivalent. Moscow ML Modules allows the declaration of applicative functors[1] as well as generative functors. An applicative version `FA` of the above functor is declared the same way, except that the formal functor argument `S : sig end` is *not* enclosed in parentheses[2]:

[1]similar to Objective Caml [Ler97] but based on our semantics in Chapter 5.

[2]this subtle syntactic distinction is in poor taste, but it avoids the addition of any new keywords and integrates well with Standard ML's syntax for functor definitions.

```
(* FA is applicative *)
functor FA S : sig end = struct datatype t = C end
structure SA1 = FA() and SA2 = FA()
val res = if true then SA1.C else SA2.C (* well-typed  *)
```

More generally, if a type in an applicative functor's body depends on a datatype or opaque type of the functor's formal argument, then the types returned by separate applications of the applicative functor will be equivalent, provided the functor is applied to equivalent type arguments:

```
(* GA is applicative, but u depends on S.t *)
functor GA S : sig type t end =
   struct datatype u = C of S.t end
structure TA1 = GA(type t = int)
structure TA2 = GA(type t = bool)
structure TA3 = GA(type t = int)
val res = if true then TA1.C 1 else TA3.C 1 (* well-typed *)
val res = if true then TA1.C 1 else TA2.C true (* ill-typed *)
```

Moscow ML's simultaneous support of both generative and applicative functors is surprisingly easy to formalise in Mini-SML. Let us assume the following syntax for (anonymous) generative and applicative functors in Mini-SML:

$$\begin{array}{rcll} \mathrm{m} & ::= & \cdots & \\ & | & \textbf{functor}\ (\mathrm{X} : \mathrm{S}) \Rightarrow \mathrm{m} & \textit{generative functor} \\ & | & \textbf{functor}\ \mathrm{X} : \mathrm{S} \Rightarrow \mathrm{m} & \textit{applicatve functor} \\ & | & \mathrm{m}\ \mathrm{m}' & \textit{functor application} \end{array}$$

We now extend the definition of higher-order semantic functors $\mathcal{F} \in \mathit{Fun}$ from Figure 5.16 to have, not just bare semantic modules, but existentially quantified modules in their range, accommodating generative types:

$$\mathcal{F} \in \mathit{Fun} \quad ::= \quad \forall P.\mathcal{M} \to \mathcal{X} \quad \textit{functor}$$

The elaboration rules for these new constructs are straightforward:

$$\frac{\mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{M} \quad P \cap \mathrm{FV}(\mathcal{C}) = \emptyset \quad \mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash \mathrm{m} : \exists Q.\mathcal{M}'}{\mathcal{C} \vdash \textbf{functor}\ (\mathrm{X} : \mathrm{S}) \Rightarrow \mathrm{m} : \exists\emptyset.\forall P.\mathcal{M} \to \exists Q.\mathcal{M}'} \qquad (\underline{H}\text{-}1)$$

($\underline{H}$-1) Similar to Rule (T-17) of the generative, first-order semantics given in Chapter 4, but the conclusion of this rule introduces an empty existential quantifier over the entire module type (the semantic functor). Note that the existential types, Q, of the functor body are simply preserved in the range of the functor signature. Contrast this behaviour with that of the applicative Rule (H-18), that skolemises the existential types in Q on the functor's type parameters, P. This module type states that the functor does not, in itself, introduce any new types, but its applications might.

$$\frac{\begin{array}{ll} \mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{M} & \vdash \mathcal{M}\ \mathbf{applicative} \\ P \cap \mathrm{FV}(\mathcal{C}) = \emptyset \quad P = \{\alpha_0^{\kappa_0}, \ldots, \alpha_{n-1}^{\kappa_{n-1}}\} & \\ \mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash \mathrm{m} : \exists Q.\mathcal{M}' & \\ Q' \cap (P \cup \mathrm{FV}(\mathcal{M}) \cup \mathrm{FV}(\exists Q.\mathcal{M}')) = \emptyset & \\ [Q'/Q] = \{\beta^\kappa \mapsto \beta^{\kappa_0 \to \cdots \kappa_{n-1} \to \kappa}\ \alpha_0 \cdots \alpha_{n-1} | \beta^\kappa \in Q\} & \\ Q' = \{\beta^{\kappa_0 \to \cdots \kappa_{n-1} \to \kappa} | \beta^\kappa \in Q\} & \end{array}}{\mathcal{C} \vdash \mathbf{functor}\ \mathrm{X} : \mathrm{S} \Rightarrow \mathrm{m} : \exists Q'.\forall P.\mathcal{M} \to \exists \emptyset.[Q'/Q]\,(\mathcal{M}')} \quad (\underline{H}\text{-}2)$$

($\underline{H}$-2) Similar to Rule (H-18) of the purely applicative semantics in Chapter 5, but the conclusion of this rule introduces an additional empty existential quantifier in the range of the generalised semantic functor to ensure the type is well-formed. This module type states that the functor itself may introduce new skolemised types, but its applications will not. The unfamiliar side condition on the type of functor argument, $\vdash \mathcal{M}$ **applicative**, is there to prevent a nasty type insecurity introduced by opaque functor signatures; its role is explained in Section 10.1.4.

$$\frac{\begin{array}{ll} \mathcal{C} \vdash \mathrm{m} : \exists P.\forall Q.\mathcal{M}' \to \mathcal{X} & \\ \mathcal{C} \vdash \mathrm{m}' : \exists P'.\mathcal{M}'' & P \cap (P' \cup \mathrm{FV}(\mathcal{M}'')) = \emptyset \\ P' \cap \mathrm{FV}(\forall Q.\mathcal{M}' \to \mathcal{X}) = \emptyset & \mathcal{M}'' \succeq \varphi\,(\mathcal{M}') \\ \mathrm{Dom}(\varphi) = Q & \varphi\,(\mathcal{X}) \equiv \exists Q'.\mathcal{M} \quad Q' \cap (P \cup P') = \emptyset \end{array}}{\mathcal{C} \vdash \mathrm{m}\ \mathrm{m}' : \exists P \cup P' \cup Q'.\mathcal{M}} \quad (\underline{H}\text{-}3)$$

(H-19) This rule combines the actions of both Rule (T-21) of Chapter 4 and Rule (H-19) of Chapter 5. Note that the functor m is an anonymous module so its type may be existentially quantified. In addition, since functors support generativity, the range of the functor is an existential module $\mathcal{X}$ (as in the generative semantics of Chapter 4), not just a bare semantic module (as in the applicative semantics of Chapter 5). To classify the application, we first eliminate the existential quantifiers in both the type of functor and the type of the argument, yielding the semantic functor $\forall Q.\mathcal{M}' \to \mathcal{X}$, and semantic module $\mathcal{M}''$. We now choose a realisation φ of the functor's type parameters Q such that $\mathcal{M}''$ enriches the realised domain $\varphi(\mathcal{M}')$. We then propagate this realisation through the range $\mathcal{X}$ of the functor yielding the result type $\varphi(\mathcal{X}) \equiv \exists Q'.\mathcal{M}$. However, because the type $\exists Q'.\mathcal{M}$ may mention the eliminated existential variables P and P', we need to ensure that they cannot escape their scope. So we re-introduce an existential quantifier that hides both P and P' in the final type of the application $\exists P \cup P' \cup Q'.\mathcal{M}$.

10.1.3 Opaque and Transparent Functor Signatures

In Moscow ML, the types of functors are specified using functor signatures. Similar to the distinction between generative and applicative functor expressions, functor signatures may be *opaque* or *transparent*. Whether a functor signature is opaque or transparent affects the interpretation of any datatype or opaque type specifications in its range signature.

Consider the *opaque* functor signature:

```
(* GO is opaque *)
signature GO =
   functor(X:sig type t val x:t end)->sig type u val y:u end
```

`GO` specifies the type of those functors that, when applied to an actual argument matching the domain signature `sig type t val x:t end`, return a result matching the range signature `sig type u val y:u end`, for some unknown implementation of the type `u` (possibly depending on `X.t`).

In Moscow ML, a *transparent* version, `GT`, of the functor signature `GO` is written in the same way, except that the formal functor argument `X:sig type t val x:t end` is *not* enclosed in parentheses:

```
(* GT is transparent *)
signature GT =
   functor X:sig type t val x:t end -> sig type u val y:u end
```

This functor signature specifies the *family* of functor types that, for a given implementation of the result type `u` (possibly depending on `X.t`), map structures matching the domain signature `sig type t val x:t end` to structures matching the range signature `sig type u val y:u end`.

In practice, when writing a functor `H` that takes a functor `F` as an argument, the choice between specifying that argument using an opaque or transparent signature will affect the amount of type information that is propagated whenever `H` is applied to an actual functor. For instance, consider the four functors:

```
functor F1 (X:sig type t val x:t end) =
   struct type u = X.t val y = X.x end

functor F2 (X:sig type t val x:t end) =
   struct type u = int val y = 1 end

functor HO(F:GO) = F(struct type t = string val x = "abc" end)
functor HT(F:GT) = F(struct type t = string val x = "abc" end)
```

Functor `F1` returns a renamed version of its argument, and functor `F2` just ignores its argument and returns the same structure regardless. The two higher-order functors `HO` and `HT` apply the supplied `F` to the same argument (in which `x` is a string), but assume, respectively, an opaque and a transparent signature for `F`.

Since functor `HO` uses the opaque signature `GO`, its formal argument `F` is assumed to return some new unknown type `u` whenever it is applied, so that the two applications of `HO` return new abstract types `RO1.u` and `RO2.u`:

```
structure RO1 = HO(F1)
structure RO2 = HO(F2)
val resO1 = if true then RO1.y else "def"  (* ill-typed *)
val resO2 = if true then RO2.y else 1  (* ill-typed *)
```

Functor `HT`, on the other hand, uses the transparent signature `GT`. This ensures that, no matter what the actual dependency of the result type `u` on the argument type `t`, `HT` may be applied to any functor `F` whose type matches `GT`, with the actual definition of `u` reflected in the result of the application. In particular, the two applications of `HT` return the same definitions for type `u` as would the substitution of `F1` and `F2` directly into the body of `HT`. That is, the types `RT1.u` and `RT2.u` are equivalent to `string` and `int`:

```
structure RT1 = HT(F1) and RT2 = HT(F2)
val resT1 = if true then RT1.y else "def"  (* well-typed *)
val resT2 = if true then RT2.y else 1  (* well-typed *)
```

Another way to look at this is that `HO`'s formal argument has a generative specification, so that its application in `HO`'s body returns a new type, while `HT`'s formal argument has an applicative specification, so that its application in `HT`'s body returns the same type as `HT`'s actual argument.

Why does Moscow have two notions of functor signature, while the Higher-Order Modules of Chapter 5 only offered one? The functor signatures of Chapter 5 are designed to capture the types of applicative functors; unfortunately, because there is no way for them to specify the existentially quantified types in the range of a semantic functor, they are not appropriate for specifying the types of generative functors. To see this, consider the generative (identity) functor:

$$\textbf{functor}\ (\mathbf{X} : \textbf{sig type t} : \mathrm{k}\ \textbf{end}) \Rightarrow (\mathbf{X} \setminus \textbf{sig type t} : \mathrm{k}\ \textbf{end})$$

which, according to our rule, has type:

$$\exists\emptyset.\forall\{\alpha\}.(\mathbf{t} = \alpha) \to \exists\{\beta\}.(\mathbf{t} = \beta)$$

Now consider the superficially appropriate functor signature:

$$\textbf{funsig}(\mathbf{X}\text{:}\textbf{sig type t} : \mathrm{k}\ \textbf{end})\textbf{sig type t} : \mathrm{k}\ \textbf{end}$$

which, according to the obvious generalisation of Rule (H-7), denotes:

$$\Lambda\{\gamma\}.\forall\{\delta\}.(\mathbf{t} = \delta) \to \exists\emptyset.(\mathbf{t} = \gamma\ \delta)$$

The scoping of β ensures that there is no realisation of δ that could make the type of the generative functor enrich the body of the realised signature. Unfortunately, if we want to use the generative functor in a higher-order way (passing it to another functor say), then we need some way to specify its type, using a different kind of functor signature. It is for these higher-order uses of generative as well as applicative functors that Moscow ML introduces the distinction between *opaque* and *transparent* functor signatures. The interpretation of the above signature assumes that the type specified in the range has a fixed functional dependency on the type parameter of the argument, which is modelled by parameterising γ on δ. The generative functor cannot match this signature because it always returns a new type β, rather than a fixed type that depends on α. The signature is transparent in the sense that matching a concrete semantic functor against this signature will reveal this dependency via the higher-order realisation of γ. For instance, the (non-generative) semantic functor $\exists\emptyset.\forall\{\alpha\}.(\mathbf{t} = \alpha) \to \exists\{\}.(\mathbf{t} = \alpha \to \alpha)$, matches this signature, and the fact that γ is realised by $\Lambda\alpha.\alpha \to \alpha$ will be apparent after matching.

Suppose that, instead of parameterising and raising the type parameter of the range, we simply used it to indicate the generativity of the functor body, so that the same syntactic signature denotes:

$$\Lambda\emptyset.\forall\{\delta\}.(\mathbf{t} = \delta) \to \exists\{\gamma\}.(\mathbf{t} = \gamma)$$

The type of the generative functor now matches this signature, because the functor's range $\exists\{\beta\}.(\mathbf{t} = \beta)$ matches the signature's range $\exists\{\gamma\}.(\mathbf{t} = \gamma)$ (eliminating the first existential quantifier, we can existentially quantify over β in $(\mathbf{t} = \beta)$ to obtain the range $\exists\{\gamma\}.(\mathbf{t} = \gamma)$). This signature is *opaque* in the sense that matching a concrete semantic functor against the signature will hide the actual realisation of any type parameters in the signature's range. For instance, the (non-generative) semantic functor $\exists\emptyset.\forall\{\alpha\}.(\mathbf{t} = \alpha) \to \exists\{\}.(\mathbf{t} = \alpha \to \alpha)$, also matches this signature, but the fact that γ is realised by $\alpha \to \alpha$ is not apparent after matching, because the realisation is hidden by the existential quantification of γ.

Let us assume the following syntax for (anonymous) generative and applicative functors in Mini-SML.

$$\begin{array}{lcll} \mathrm{m} & ::= & \cdots & \\ & | & \textbf{functor}\ (\mathrm{X}{:}\mathrm{S}) \to \mathrm{S}' & \textit{opaque functor signature} \\ & | & \textbf{functor}\ \mathrm{X}{:}\mathrm{S} \to \mathrm{S}' & \textit{transparent functor signature} \end{array}$$

As before, the denotations of opaque and transparent functor signatures are just parameterised semantic functors. The difference between the two constructs is the way in which they treat the parameters of the range signature S′: an opaque functor signature merely uses these parameters to determine the existential quantifier of the range of the semantic functor; the transparent functor signature uses the parameters to determine the higher-order parameters of the entire signature, capturing any functional dependencies of the type components in the range on the type components in the domain.

The elaboration rules for these constructs are straightforward:

$$\frac{\mathcal{C} \vdash \mathrm{S} \rhd \Lambda P.\mathcal{M} \quad P \cap \mathrm{FV}(\mathcal{C}) = \emptyset \quad \mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash \mathrm{S}' \rhd \Lambda Q.\mathcal{M}'}{\mathcal{C} \vdash \textbf{functor}\ (\mathrm{X}{:}\mathrm{S}) \to \mathrm{S}' \rhd \Lambda\emptyset.\forall P.\mathcal{M} \to \exists Q.\mathcal{M}'} \qquad (\underline{H}\text{-}4)$$

($\underline{H}$-4) The rule is similar to Rule ($\underline{H}$-1) that relates generative functors to their types. Unlike Rule (H-7) of the purely applicative semantics in

Chapter 5, this rule introduces an empty set of parameters over the entire module type (the semantic functor). Note that the parameters $\mathcal{Q}$ of the range signature, S, simply determine the existentially quantified variables of the semantic functor's range; in particular, they are *not* replaced by new higher-order variables that encode dependencies on the type parameters in P(the behaviour of Rule (H-7)).

$$\frac{\begin{array}{l} \mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{M} \\ P \cap \mathrm{FV}(\mathcal{C}) = \emptyset \quad P = \{\alpha_0^{\kappa_0}, \ldots, \alpha_{n-1}^{\kappa_{n-1}}\} \\ \mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash \mathrm{S}' \triangleright \Lambda Q.\mathcal{M}' \\ Q' \cap (P \cup \mathrm{FV}(\mathcal{M}) \cup \mathrm{FV}(\Lambda Q.\mathcal{M}')) = \emptyset \\ [Q'/Q] = \{\beta^\kappa \mapsto \beta^{\kappa_0 \to \cdots \kappa_{n-1} \to \kappa}\ \alpha_0 \cdots \alpha_{n-1} | \beta^\kappa \in Q\} \\ Q' = \{\beta^{\kappa_0 \to \cdots \kappa_{n-1} \to \kappa} | \beta^\kappa \in Q\} \end{array}}{\mathcal{C} \vdash \textbf{functor}\ \mathrm{X}{:}\mathrm{S} \to \mathrm{S}' \triangleright \Lambda Q'.\forall P.\mathcal{M} \to \exists\emptyset.[Q'/Q]\,(\mathcal{M}')} \qquad (\underline{H}\text{-}5)$$

($\underline{H}$-5) This rule is similar to Rule (H-7), for functors in the purely applicative semantics, but introduces an additional empty existential quantifier in the range of the semantic functor to ensure the type is well-formed.

All that remains is to extend the definition of the enrichment relation for higher-order Mini-SML to cater for our generalised notion of semantic functor with existentially quantified range. This, too, is straightforward. Informally, we modify the relevant clauses of Specification 5.12 as follows:

Specification 10.1 (Enrichment).

$\boxed{\mathcal{F} \succeq \mathcal{F}'}$ *Given two functors $\mathcal{F}$ and $\mathcal{F}'$, $\mathcal{F}$* enriches *$\mathcal{F}'$, written $\mathcal{F} \succeq \mathcal{F}'$, if, and only if, every instance of $\mathcal{F}'$ is an instance of $\mathcal{F}$, i.e. for any modules $\mathcal{M}$ and $\mathcal{X}$, $\mathcal{F}' > \mathcal{M} \to \mathcal{X}$ implies $\mathcal{F} > \mathcal{M} \to \mathcal{X}$.*

$\boxed{\mathcal{X} \succeq \mathcal{X}'}$ *Given two existential modules $\mathcal{X}$ and $\mathcal{X}'$, $\mathcal{X}$* enriches *$\mathcal{X}'$, written $\mathcal{X} \succeq \mathcal{X}'$, if, and only if, $\mathcal{X} \equiv \exists P.\mathcal{M}$ and $\mathcal{X}' \equiv \exists P'.\mathcal{M}'$, and for some realisation φ with* $\mathrm{Dom}(\varphi) = P'$, *$\mathcal{M} \succeq \varphi(\mathcal{M}')$, where $P \cap \mathrm{FV}(\mathcal{X}') = \emptyset$.*

$\boxed{\mathcal{F} > \mathcal{M} \to \mathcal{X}}$ *A* functor instance *$\mathcal{M} \to \mathcal{X}$ is the type of a monomorphic function on modules.*

Given functor $\mathcal{F} \equiv \forall P.\mathcal{M}_P \to \mathcal{X}_P$, $\mathcal{M} \to \mathcal{X}$ is an instance *of $\mathcal{F}$, written $\mathcal{F} > \mathcal{M} \to \mathcal{X}$, if, and only if, for some realisation φ with* $\mathrm{Dom}(\varphi) = P$, *$\mathcal{M} \succeq \varphi(\mathcal{M}_P)$ and $\varphi(\mathcal{X}_P) \succeq \mathcal{X}$.*

The enrichment relation on existential modules types merely requires that, eliminating the existential quantifier from the more general module type $\exists P.\mathcal{M}$, there is some realisation φ of the existential variables P' of the second module type $\exists P'.\mathcal{M}'$ that allows the more general type $\mathcal{M}$ to enrich the realised, less general type $\varphi(\mathcal{M}')$.

Just like Specification 5.12 this intuitive specification cannot serve as a proper definition of the enrichment relations. Instead, we can define them formally as follows:

Definition 10.2 (Enrichment).

The enrichment relations between structures, functors and modules are defined as the least relations $\_ \succeq \_ \in \mathit{Str} \times \mathit{Str}$, $\_ \succeq \_ \in \mathit{Fun} \times \mathit{Fun}$, and $\_ \succeq \_ \in \mathit{Mod} \times \mathit{Mod}$ closed under the rules in Figure 10.1.

Generalising the higher-order signature matching algorithm (Definition 5.25) to handle generative functors requires a simple modification, shown in Figures 10.2 and 10.3. The only interesting new rule of this generalised matching algorithm is Rule ($\underline{M}$-9). The first side condition on N ensures that the existentially quantified variables of the lesser existential module are treated as fresh hypothetical types. In particular, these types are distinct from the free (and bound) variables of the more general existential module. Note that the realisation φ', returned by the recursive call in the premise, is allowed to realise variables in M, (as well as any others variables not in $P \cup R \cup N$). In φ, the realisation of these existentially quantified variables is hidden by restricting the domain of φ' outside M. The second side condition on N ensures that the hypothetical types cannot escape their scope by appearing in the range of φ. It does not, however, prevent them from occurring in the range of φ', as they may occur in the realisation of the existential variables in M (which is correct because M and N are quantified at the same position).

Structure Enrichment $\boxed{\mathcal{S} \succeq \mathcal{S}'}$

$$\frac{\begin{array}{l}\mathrm{Dom}(\mathcal{S}) \supseteq \mathrm{Dom}(\mathcal{S}') \\ \forall \mathrm{t} \in \mathrm{Dom}(\mathcal{S}').\mathcal{S}(\mathrm{t}) = \mathcal{S}'(\mathrm{t}) \\ \forall \mathrm{x} \in \mathrm{Dom}(\mathcal{S}').\mathcal{S}(\mathrm{x}) \succeq \mathcal{S}'(\mathrm{x}) \\ \forall \mathrm{X} \in \mathrm{Dom}(\mathcal{S}').\mathcal{S}(\mathrm{X}) \succeq \mathcal{S}'(\mathrm{X})\end{array}}{\mathcal{S} \succeq \mathcal{S}'} \qquad (\underline{\succeq}\text{-1})$$

Functor Enrichment $\boxed{\mathcal{F} \succeq \mathcal{F}'}$

$$\frac{\begin{array}{ll}\mathcal{M}_Q \succeq \varphi(\mathcal{M}_P) & \varphi(\mathcal{X}_P) \succeq \mathcal{X}_Q \\ \mathrm{Dom}(\varphi) = P & Q \cap \mathrm{FV}(\forall P.\mathcal{M}_P \to \mathcal{X}_P) = \emptyset\end{array}}{\forall P.\mathcal{M}_P \to \mathcal{X}_P \succeq \forall Q.\mathcal{M}_Q \to \mathcal{X}_Q} \qquad (\underline{\succeq}\text{-2})$$

Module Enrichment $\boxed{\mathcal{M} \succeq \mathcal{M}'}$

$$\frac{\mathcal{S} \succeq \mathcal{S}'}{\mathcal{S} \succeq \mathcal{S}'} \qquad (\underline{\succeq}\text{-3})$$

$$\frac{\mathcal{F} \succeq \mathcal{F}'}{\mathcal{F} \succeq \mathcal{F}'} \qquad (\underline{\succeq}\text{-4})$$

Existential Module Enrichment $\boxed{\mathcal{X} \succeq \mathcal{X}'}$

$$\frac{\mathcal{M}_P \succeq \varphi(Q) \quad \mathrm{Dom}(\varphi) = Q \quad P \cap \mathrm{FV}(\exists Q.\mathcal{M}_Q) = \emptyset}{\exists P.\mathcal{M}_P \succeq \exists Q.\mathcal{M}_Q} \qquad (\underline{\succeq}\text{-5})$$

Figure 10.1: An inductive definition of enrichment for Higher-Order Modules with generative functors.

Structure Matching $\boxed{\forall P.\forall R \vdash \mathcal{S} \succeq \mathcal{S}' \downarrow \varphi}$

$$\frac{}{\forall P.\forall R \vdash \mathcal{S} \succeq \epsilon_{\mathcal{S}} \downarrow \emptyset} \qquad (\underline{M}\text{-}1)$$

$$\frac{\mathrm{t} \in \mathrm{Dom}(\mathcal{S}) \quad \mathcal{S}(\mathrm{t}) = \tau \quad \forall P.\forall R \vdash \mathcal{S} \succeq \mathcal{S}' \downarrow \varphi}{\forall P.\forall R \vdash \mathcal{S} \succeq \mathrm{t} = \tau, \mathcal{S}' \downarrow \varphi} \qquad (\underline{M}\text{-}2)$$

$$\frac{\begin{array}{l} \alpha \notin P \cup R \\ \mathrm{t} \in \mathrm{Dom}(\mathcal{S}) \\ \mathcal{S}(\mathrm{t}), (\alpha\ \beta_0 \cdots \beta_{n-1}) \in \mathit{Typ}^{\kappa} \\ \mathrm{FV}(\mathcal{S}(\mathrm{t})) \cap R \subseteq \{\beta_i \mid i \in [n]\} \\ \forall P.\forall R \vdash \mathcal{S} \succeq [\Lambda\beta_0 \cdots \beta_{n-1}.\mathcal{S}(\mathrm{t})/\alpha]\,(\mathcal{S}') \downarrow \varphi \end{array}}{\forall P.\forall R \vdash \mathcal{S} \succeq \mathrm{t} = \alpha\ \beta_0 \cdots \beta_{n-1}, \mathcal{S}' \downarrow ([\Lambda\beta_0 \cdots \beta_{n-1}.\mathcal{S}(\mathrm{t})/\alpha] \mid \varphi)} \qquad (\underline{M}\text{-}3)$$

$$\frac{\mathrm{x} \in \mathrm{Dom}(\mathcal{S}) \quad \mathcal{S}(\mathrm{x}) \succeq v \quad \forall P.\forall R \vdash \mathcal{S} \succeq \mathcal{S}' \downarrow \varphi}{\forall P.\forall R \vdash \mathcal{S} \succeq \mathrm{x} : v, \mathcal{S}' \downarrow \varphi} \qquad (\underline{M}\text{-}4)$$

$$\frac{\mathrm{X} \in \mathrm{Dom}(\mathcal{S}) \quad \forall P.\forall R \vdash \mathcal{S}(\mathrm{X}) \succeq \mathcal{M} \downarrow \varphi \quad \forall P.\forall R \vdash \mathcal{S} \succeq \varphi\,(\mathcal{S}') \downarrow \varphi'}{\forall P.\forall R \vdash \mathcal{S} \succeq \mathrm{X} : \mathcal{M}, \mathcal{S}' \downarrow (\varphi \mid \varphi')} \qquad (\underline{M}\text{-}5)$$

Figure 10.2: A generalised algorithm for matching, suitable for both transparent and opaque functor signatures. Subject to certain constraints, we have $\forall P.\forall R \vdash \mathcal{O} \succeq \mathcal{O}' \downarrow \varphi$ if, and only if, $\mathcal{O} \succeq \varphi\,(\mathcal{O}')$.

Functor Matching $\boxed{\forall P.\forall R \vdash \mathcal{F} \succeq \mathcal{F}' \downarrow \varphi}$

$$\frac{\begin{array}{l} N \cap (P \cup R \cup M) = \emptyset \\ M \cap (P \cup R) = \emptyset \\ \forall P \cup R \cup M.\forall\emptyset \vdash \mathcal{M}_M \succeq \mathcal{M}_N \downarrow \varphi' \\ \forall P.\forall R \cup M \vdash \varphi'(\mathcal{X}_N) \succeq \mathcal{X}_M \downarrow \varphi \end{array}}{\forall P.\forall R \vdash \forall N.\mathcal{M}_N \to \mathcal{X}_N \succeq \forall M.\mathcal{M}_M \to \mathcal{X}_M \downarrow \varphi} \quad (\underline{M}\text{-6})$$

Module Matching $\boxed{\forall P.\forall R \vdash \mathcal{M} \succeq \mathcal{M}' \downarrow \varphi}$

$$\frac{\forall P.\forall R \vdash \mathcal{S} \succeq \mathcal{S}' \downarrow \varphi}{\forall P.\forall R \vdash \mathcal{S} \succeq \mathcal{S}' \downarrow \varphi} \quad (\underline{M}\text{-7}) \qquad \frac{\forall P.\forall R \vdash \mathcal{F} \succeq \mathcal{F}' \downarrow \varphi}{\forall P.\forall R \vdash \mathcal{F} \succeq \mathcal{F}' \downarrow \varphi} \quad (\underline{M}\text{-8})$$

Existential Module Matching $\boxed{\forall P.\forall R \vdash \mathcal{X} \succeq \mathcal{X}' \downarrow \varphi}$

$$\frac{\begin{array}{ll} N \cap (P \cup R \cup M) = \emptyset & \\ M \cap (P \cup R) = \emptyset & \forall P \cup N.\forall R \vdash \mathcal{M}_N \succeq \mathcal{M}_M \downarrow \varphi' \\ \varphi = \{\alpha \mapsto \varphi'(\alpha) \,|\alpha \in \mathrm{Dom}(\varphi') \setminus M\} & N \cap \bigcup_{\alpha \in \mathrm{Dom}(\varphi)} \mathrm{FV}(\varphi(\alpha)) = \emptyset \end{array}}{\forall P.\forall R \vdash \exists N.\mathcal{M}_N \succeq \exists M.\mathcal{M}_M \downarrow \varphi}$$

$(\underline{M}\text{-9})$

Figure 10.3: (continuation of Figure 10.2.)

10.1.4 Closing the Loopholes

It was recently discovered that the naive combination of generative and applicative functors, as implemented in Moscow ML 2.00 is, in fact, unsound [DCH02]. A simple counter-example, adapted from [DCH02] but similar to the one in Figure 7.12, reveals the type loophole:

```
signature S = sig type t val x: t val f: t -> int end

(* App is an applicative functor
 ... with a generative functor argument F *)
functor App F:functor (X:sig end)->S = F()

functor G (X:sig end) = struct
   type t = int
   val x = 0
   val f = fn y => y+1
end
functor H (X: sig end) = struct
   type t = int -> int
   val x = fn x => x
   val f = fn y => y 1
end
structure X = App(G)
structure Y = App(H)

(* since App is applicative X.t = Y.t *)
val wrong = Y.f (X.x) (* applies 0 to 1, a run-time error *)
```

The root of the problem is the applicative functor App. The introduction rule for applicative functors skolemises any existentials introduced in the functor's body on the type parameters to the functor. This is sound provided the witnesses to these existentials are known functions of the functor's type parameters. In particular, they must not have a dynamic dependency on the value of the functor's argument. In the counter-example, the witness to the functor body's existential type is determined by the formal argument of App. But the formal argument is specified to be a functor F with an *opaque* functor signature:

$$\forall\emptyset.\epsilon_{\mathcal{S}} \to \exists\{\alpha\}.(\mathbf{t} = \alpha, \mathbf{x} : \alpha, \mathbf{f} : \alpha \to \mathbf{int})$$

Applicative Structures $\boxed{\vdash \mathcal{S}\ \textbf{applicative}}$

$$\frac{\forall \mathrm{X} \in \mathrm{Dom}(\mathcal{S}).\vdash \mathcal{S}(\mathrm{X})\ \textbf{applicative}}{\vdash \mathcal{S}\ \textbf{applicative}} \qquad (\textit{App}\text{-}1)$$

Applicative Functors $\boxed{\vdash \mathcal{F}\ \textbf{applicative}}$

$$\frac{\vdash \mathcal{M}'\ \textbf{applicative}}{\vdash \forall P.\mathcal{M} \to \exists\emptyset.\mathcal{M}'\ \textbf{applicative}} \qquad (\textit{App}\text{-}2)$$

Applicative Modules $\boxed{\vdash \mathcal{M}\ \textbf{applicative}}$

$$\frac{\vdash \mathcal{S}\ \textbf{applicative}}{\vdash \mathcal{S}\ \textbf{applicative}} \qquad (\textit{App}\text{-}3)$$

$$\frac{\vdash \mathcal{F}\ \textbf{applicative}}{\vdash \mathcal{F}\ \textbf{applicative}} \qquad (\textit{App}\text{-}4)$$

Figure 10.4: Applicative module types.

The type component returned by `F` is generative and bound in the range of `F`. This means that it is unknown and allowed to vary with each application of `F`. According to our rules, the body of `App` receives type:

$$\exists\{\beta\}.(\mathbf{t} = \beta, \mathbf{x} : \beta, \mathbf{f} : \beta \to \mathbf{int})$$

But skolemising the new type β on the (empty) set of parameters in the argument signature does not take β's dynamic dependency on `F` into account, leading to this unsound type for `App`:

$$\begin{gathered}\forall\emptyset.(\forall\emptyset.\epsilon_{\mathcal{S}} \to \exists\{\alpha\}.(\mathbf{t} = \alpha, \mathbf{x} : \alpha, \mathbf{f} : \alpha \to \mathbf{int})) \to \\ \exists\emptyset.(\mathbf{t} = \beta, \mathbf{x} : \beta, \mathbf{f} : \beta \to \mathbf{int})\end{gathered}$$

This type states that each application of `App` returns a constant type for **t**, regardless of the actual argument. This is a lie: in the counter-example, the two distinct applications of `App` yield radically different underlying types for **t** and these must be kept distinct by the type system.

This kind of unsoundness never arises in the purely applicative semantics of Chapter 5, because the less expressive functor signatures considered there can only abstract over, but never hide, a functor's argument-result type dependencies.

So how do can we avoid this unsoundness in Moscow ML? One draconian solution, suggested in [DCH02], is to rule out opaque functor signatures altogether, but this demotes Standard ML's generative functors to second-class citizens. A less restrictive alternative, adopted here and in a future release of Moscow ML, is to place an additional restriction on the introduction rule for applicative functors (Rule $\underline{H}$-2): the type of the formal argument must be *applicative*, according to the definition in Figure 10.4. The restriction is designed to prevent an applicative functor from taking a formal argument, that, either directly by applicaton, or indirectly by projections and/or curried application, may be used in a generative manner. Intuitively, any (higher-order) argument of the functor must be strictly non-generative in its positive positions. The restriction avoids the above counter-example by ruling out the definition of `App`. Note that the introduction rule for generative functors remains unrestricted. The proof that this side-condition is sufficient to ensure soundness is left to future work.

10.1.5 Generalised Projections

In SML, one can only project a type component from a named module, forcing the evaluation of that module binding. Moscow ML additionally supports generalised type projections of the form

longtycon **where** strbind

Here, longtycon is a type path and strbind is a structure binding local to that path. This makes it possible to refer directly to a type returned by an applicative functor within another type expression, which is useful for expressing sharing constraints:

```
signature S = sig functor F: functor X:sig type t end ->
                                sig type u end
                  type v = X.u where X = F(type t = int)
                           (* X is local to X.u *)
              end
```

The local binding has no run-time effect and is only elaborated at compile-time for its type information. This new phrase is nothing more than a syntactic variant of the generalised dot notation for types, m.t, introduced in Section 5.2.3, but fits better with the concrete grammar of SML. Moscow ML does not add any new syntax for generalised projections of value or module components because these can already be expressed using existing constructs.

10.1.6 First-class Modules

In Moscow ML Modules, a structure or functor value can be packaged as a Core value of *package type*, manipulated as any other first-class value of the Core, and finally unpacked to access the enclosed structure or functor:

```
signature NAT = sig type nat
                    val Z:nat
                    val S:nat -> nat
                    val plus: nat -> nat -> nat
                end
```

```
structure SafeNat = (* unlimited range but slow *)
  struct datatype nat = Z | S of nat
         fun plus Z m = m
           | plus (S n) m = S (plus n m)
  end

structure FastNat = (* limited range but fast *)
  struct type nat = int
         val Z = 0
         fun S n = n + 1
         fun plus n m = n + m
  end

type natpack = [ NAT ]                      (* package type *)

val safeNat = [ structure SafeNat as NAT ] (* packing *)
val fastNat = [ structure FastNat as NAT ]

structure Nat as NAT =                      (* unpacking *)
      if (913 mod 7 = 5) then safeNat else fastNat
val natlist = [safeNat,fastNat] : [ NAT ] list
```

A functor may be packed using the similar Core expression `[ functor` *modexp* `as` *sigexp* `]` and unpacked using the functor binding `functor` *funid* `as` *sigexp* `=` *exp*.

Package type equivalence is determined by structure, not name, so the following package types are equivalent:

```
[sig type t val x: t type u = t val y: u end]
[sig type u val x: u type t = u val y: t end]
```

because the signatures are equivalent (every structure that matches one also matches the other).

For type soundness reasons, a package may not be unpacked in the body of a functor (although it may be unpacked within a Core expression occurring in that body):

```
(* illegal *)
functor Fail (val nat : [ NAT ]) =
  struct structure Nat as NAT = nat end
```

```
(* legal *)
functor Ok (val nat : [ NAT ]) =
  struct val x = let structure Nat as NAT = nat in nat end
  end
```

How do Moscow's first-class modules relate to the proposal in Chapter 7? For syntactic reasons, Moscow uses separate Core expressions for packing structures and functors, respectively, but the semantics of these constructs is essentially the same as that of our single **pack** m **as** S construct. A more significant departure is our use of a special definition (or declaration in SML terminology), rather than a new Core expression, to eliminate package types. The construct proposed in this thesis, **open** e **as** X : S **in** e′, is inconvenient because it only allows a package value to be opened within the scope of a Core expression, and not, for instance, at top-level or within a structure body. In Moscow ML, we relax this restriction by omitting the **open** construct, introducing in its place two new forms of structure and functor definition. These definitions are used to eliminate package types from Core expressions, by binding module identifiers to their contents. They may appear wherever ordinary SML module definitions can. The declaration of `Nat` above, illustrates the convenience of this construct for expressing dynamic definitions: without it, we would have to express the rest of the top-level program as the body of a Mini-SML **open**- expression.

The alternative elimination syntax can be formalised in Mini-SML by adding a new module definition phrase,

$$\mathrm{b} ::= \ \ldots \ \mid \ \textbf{module}\ \mathrm{X}\ \textbf{as}\ \mathrm{S} = \mathrm{e}; \mathrm{b}\ ,$$

with classification rule:

$$\frac{\begin{array}{ll} \mathcal{C} \vdash \mathrm{e} : <\exists P.\mathcal{M}> & P \cap \mathrm{FV}(\mathcal{C}) = \emptyset \\ \mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash \mathrm{b} : \exists P'.\mathcal{S} & \\ P' \cap (P \cup \mathrm{FV}(\mathcal{M})) = \emptyset & \mathrm{X} \notin \mathrm{Dom}(\mathcal{S}) \end{array}}{\mathcal{C} \vdash \textbf{module}\ \mathrm{X}\ \textbf{as}\ \mathrm{S} = \mathrm{e}; \mathrm{b} : \exists P \cup P'.\mathrm{X} : \mathcal{M}, \mathcal{S}}$$

Our **open** construct is subsumed by the ordinary SML **let** expression, which can be captured in Mini-SML by adding a new Core expression:

$$\mathrm{e} ::= \ \ldots \ \mid \ \textbf{let}\ \mathrm{b}\ \textbf{in}\ \mathrm{e}\ \textbf{end}$$

Compared with the simpler **let** x = e′ **in** e expression of Chapter 3, this more general expression allows an arbitrary sequence of definitions, b, to be declared within the expression e. In particular, b may contain bindings of types and modules, not just values. The corresponding classification rule can be formalised in Mini-SML as:

$$\frac{\begin{array}{ll}\mathcal{C} \vdash \mathrm{b} : \exists P.\mathcal{S} & P \cap \mathrm{FV}(\mathcal{C}) = \emptyset \\ \mathcal{C}, \mathcal{S} \vdash \mathrm{e} : u & P \cap \mathrm{FV}(u) = \emptyset\end{array}}{\mathcal{C} \vdash \textbf{let}\ \mathrm{b}\ \textbf{in}\ \mathrm{e}\ \textbf{end} : u}$$

where $\mathcal{C}, \mathcal{S}$ is the obvious pointwise extension of $\mathcal{C}$ by the components of $\mathcal{S}$. In SML, the static semantics for such expressions, like Rule (P-3), already ensures that the type u of e cannot depend on any abstract type introduced in b, which is prevented here by the side-condition $P \cap \mathrm{FV}(u) = \emptyset$. Clearly, **open** e **as** X : S **in** e$'$ is definable as **let module** X **as** S = e **in** e$'$ **end**.

Unlike our original **open** construct, Moscow's more liberal syntax can violate the invariant needed to support *applicative* functors. Recall that the required property is that the abstract types returned by a functor should depend only on its type arguments and not the value of its term argument. A small counter-example, similar to the one in Figure 7.12, demonstrates how the new construct could break this invariant:

```
signature S = sig type t  val x: t  val y: t -> int end

functor F A:sig val b: bool end = struct
  structure X as S = if A.b
     then [structure struct type t = int
                            val x = 0
                            val y = fn x:t => x
                        end as S]
     else [structure struct type t = int -> int
                            val x = fn x:int => x
                            val y = fn f:t => f 1
                        end as S]
end (* X.t depends on the value of A.b *)
structure Y = F(struct val b = true end)
structure Z = F(struct val b = false end)
(* since F is applicative, Y.X.t = Z.X.t *)
val z = Z.X.y (Y.X.x)  (* applies 0 to 1, a run-time error *)
```

To restore the invariant, Moscow ML imposes a syntactic restriction: a package eliminating module definition cannot appear in a functor body (unless the definition is enclosed by an inner Core **let**-expression). This rules out the previous counter-example and others like it.

The syntactic restriction must be applied to all functors, even generative ones. This is because a generative functor can always be turned into an applicative one by an η-expansion:

```
(* G is generative *)
functor G (A:sig val b: bool end) = struct
  structure X as S = if A.b
     then [structure struct type t = int
                            val x = 0
                            val y = fn x:t => x
                     end as S]
     else [structure struct type t = int -> int
                            val x = fn x:int => x
                            val y = fn f:t => f 1
              end as S]
end (* X.t depends on the value of A.b *)
functor F A:sig val b:bool end = G(A)
(* F is an applicative η-expansion of G *)
structure Y = F(struct val b = true end)
structure Z = F(struct val b = false end)
(* since F is applicative Y.X.t = Z.X.t *)
val wrong = Z.X.y Y.X.x  (* applies 0 to 1, a run-time error *)
```

The syntactic restriction rejects the (otherwise sound) definition of `G`, to prevent the subsequent unsound definition of `F`.

Note that the restriction only applies to functor bodies, not top-level structures or their substructures, which may still contain package-eliminating module bindings. Recalling the discussion in Sections 7.4 and 7.4.1, intuitively, the reason it is sound to admit top-level package-elimination is that any existential types introduced at top-level will never be skolemised on the type parameters of an outer functor argument (since there can be no such outer functor). For this reason, it is perfectly legal to allow the realisations of these variables to depend on the dynamic as well as the static interpretation of the context.

10.1.7 Recursive Structures and Signatures

SML Modules does not support the *recursive* definition of modules. For instance, two structures `Even` and `Odd` cannot refer to each other, nor can the body of a single structure depend on its own definition. In Moscow ML Modules, structures *can* be defined recursively:

```
structure S =
 rec(X:sig structure Odd : sig val test : int -> bool end end)
 struct structure Even = struct fun test 0 = true
                                  | test n = X.Odd.test (n-1)
                         end
        structure Odd  = struct fun test 0 = false
                                  | test n = Even.test (n-1)
                         end
 end
```

Here, `X` is a forward declaration of the structure's body that allows the Core value `Even.test` to refer to `X.Odd.test` before it has been defined. In this new form of structure expression, **rec**(X : S)s, X is a forward declared structure, S is its signature, and s is the actual body of the structure. The body of a recursive structure must enrich the signature of the forward declaration; any opaque type or datatype specified in the signature must be implemented in the body by *copying* it using a forward reference:

```
(* well-typed *)
structure Ok =
   rec(X:sig datatype t = C
             type u
             type v = int
         end)
   struct datatype t = datatype X.t
          type u = X.u
          type v = int
   end

(* ill-typed *)
structure Fail =
   rec(X:sig datatype t = C
             type u
             type v = int
         end)
   struct datatype t = C
          type u = int
          type v = int
   end
```

In the dynamic semantics, the body of a recursive structure is evaluated eagerly, under the initial assumption that its forward declaration is undefined. If evaluation of the body reaches a value, the assumption is updated with that value. Attempting to evaluate the forward reference of a recursive structure before its body has been fully evaluated raises the exception `Bind`:

```
structure Fail = rec(X:sig end)X  (* raises Bind *)
structure Fail = rec(X:sig val x: int end)
                 struct val x = X.x end (* raises Bind *)


structure Ok = rec (X:sig val f: int -> int end)
               struct fun f n = X.f n  end  (* terminates *)
val res = Ok.f n (* loops *)
```

On their own, recursive structures cannot be used to declare mutually recursive datatypes that span module boundaries. For this purpose, Moscow ML also supports *recursive signatures*:

```
signature REC =
   rec(X: sig structure Odd: sig type t end end)
      sig structure Even: sig datatype t = Zero
                                         | Succ of X.Odd.t
                          end
          structure  Odd: sig datatype t = One
                                         | Succ of Even.t
                          end
      end
```

Here, `X` is a forward declaration of a structure implementing the body of the signature that allows the specification of `Even.t` to refer to the type `X.Odd.t` before it has been fully specified. In this new form of signature expression, $\mathbf{rec}(\mathrm{X} : \mathrm{S})\mathrm{S}'$, X is a forward declared structure, S is its signature, and S' is the body of the signature, which may refer to its own components via X. In a recursive signature, the body of the signature must match the forward declaration and specify an implementation for any opaque types or datatypes declared within the forward specification.

Once a recursive signature has been defined, it can be used to implement a recursive structure that does contain datatypes spanning module boundaries. To illustrate, we can use the signature `REC` to define the structure `T`:

```
structure T =
  rec(X:REC)
  struct structure Even = struct datatype t = datatype X.Even.t
                                 fun succ Zero = X.Odd.One
                                   | succ E = X.Odd.Succ E
                          end
         structure Odd  = struct datatype t = datatype X.Odd.t
                                 fun succ O = Even.Succ O
                          end
  end
```

Note that `T.Even.t` and `T.Odd.t` are mutually recursive, and the constructors of the datatype defined in one structure are visible to the other, supporting the definitions of the two successor functions.

The formal details of recursive modules take us beyond the scope of this monograph. An adequate exposition requires a proper formalisation of SML's datatype definitions and specifications as well as a presentation of the dynamic semantics of recursive structure expressions. The interested reader is referred to [Rus01], in which the extension is formulated for a variant of Mini-SML. A notable feature of this work is that it builds directly on the concepts and style of static semantics presented in this thesis. In Moscow ML, this extension has the pleasant side-effect of extending the language with (explicitly declared) polymorphic recursion, a feature missing from SML. It also supports the construction of recursive values other than Core functions (such as records, which are useful for encoding "objects").

10.1.8 Relaxations of Other Standard ML Restrictions

In SML, functors and signatures may only be declared at top-level, and structures may only be declared at top-level and within structures. None of these may be declared within Core **let**-expressions. Moscow ML removes these restrictions so that functors, signatures and structures may be declared anywhere, which is particularly useful when programming with first-class modules.

In SML, every parameterised type definition, and every type scheme occurring within a signature, must be closed: it must not mention any (simple) type variables that are not explicitly listed as type parameters. Moscow ML does not impose this restriction, and allows free simple type variables, provided they are (explicitly or implicitly) bound in an enclosing scope. Again, this is useful when programming with first-class modules.

```
(* illegal, since 'a is not in scope *)
type t = 'a -> 'a

(* legal Moscow ML, but illegal Standard ML *)
fun f (x:'a) = let type t = 'a * 'a
               in (x,x):t
               end

(* legal Moscow ML *)
type 'a stackpack =
    [ sig
          (* 'a occurs free in this type binding *)
          type stack = 'a list
          (* 'a occurs free in this type scheme  *)
          val push : 'a -> stack -> stack
      end ]
```

10.1.9 Miscellaneous Static Semantic Issues

Scaling the extensions described here to SML involves revising some other features of the SML static semantics, including the formalisation of datatypes and equality types in the presence of higher-order type variables and functions (introduced by applicative functors and transparent functor signatures), extending the definition of implicit scoping of explicit type parameters, and enlarging the class of non-expansive Core expressions to support non-expansive package expressions. The details of these modifications are not surprising but require too much preparatory material to present here, in the simple format of an extension to our Mini-SML semantics. The interested reader should contact the author for more details and a copy of the original design document [Rus00c].

10.1.10 Compiling Modules

In Moscow ML, we adopt a conventional compilation scheme and compile a structure whose type contains n named values to a vector of n anonymous values identified by their position (type components have no run-time representation). More elaborate schemes are possible [Els99] but this one is reasonably simple and efficient. The vector position of each named component within a structure is calculated at compile-time; because enrichment can remove some components from a structure's type, a coercion is inserted wherever the enrichment relation occurs in the typing derivation (e.g. in instances of Rule (P-2)). These coercions are constructed by induction on the structure of the types related by enrichment. The vector representation thus offers efficient component access at the cost of some run-time coercions. It is a simple optimisation to prune those coercions that turn out to be the identity. When compiling ordinary SML Modules, it is sufficient to assign vector positions to a structure's components according to their textual order of definition in the structure body and to assume vector positions for a signature's components according to their textual order of specification in the signature body. However, in the presence of first-class structures, one must additionally ensure that packaged structures with equivalent types share the same vector representation (even if the components of the structures were declared or specified in a different textual order). In our Moscow ML implementation, we assume a total ordering on term identifiers, use it to sort the domain of each structure's type and finally assign vector positions to components according to the relative order of their names. Since equivalent package types have equivalent domains (a property that extends to their sub-structures) their values receive compatible representations. Adopting this canonical representation has no run-time cost: in fact, it can improve the compilation of ordinary SML programs by revealing more optimisation opportunities (identity coercions) than the non-canonical scheme. To illustrate, the array example of Section 7.3 requires no run-time coercions at all, as all the intermediate structure expressions have exactly the same domain as the signature `Array`.

Functors, whether generative or applicative, are simply compiled as functions in the untyped intermediate language of Moscow ML.

Bibliography

[Apo93] María Virginia Aponte. Extending record typing to type parametric modules with sharing. In *Proc. 20th Symp. Principles of Prog. Lang.*, pages 465–478. ACM Press, 1993. 2.3.4

[AT 93] AT & T Bell Laboratories. *Standard ML of New Jersey, User's Guide*, 0.93 edition, February 1993. 2.3.4

[Bis95] Sandip K. Biswas. Higher-order functors with transparent signatures. In *Proc. 22nd Symp. Principles of Prog. Lang.*, pages 154—163. ACM Press, 1995. 1.2, 2.3.4, 5, 5.1, 5.2.1, 5.2.2, 10, 5.8, 9.1

[BL84] R. M. Burstall and B. Lampson. A kernel language for abstract data types and modules. In *Semantics of Data Types*, number 173 in LNCS. Springer-Verlag, 1984. 2.3.1

[BL88] R. M. Burstall and B. Lampson. Pebble, a kernel language for modules and abstract data types. *Information and Computation*, 76:278–346, 1988. 2.3.1

[Car88a] Luca Cardelli. A Quest Preview. Technical report, Digital Equipment Corporation, Systems Research Center, June 1988. 2.3.1

[Car88b] Luca Cardelli. Phase Distinctions in Type Theory. Unpublished manuscript available from `http://www.luca.demon.co.uk`, 1988. 2.2.2, 2.2.5

[Car89] Luca Cardelli. Typeful Programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, IFIP State of the Art Reports Series. Springer-Verlag, February 1989. 2.3.1

[Car91] Luca Cardelli. *The Quest Language and System.* Digital Equipment Corporation, Systems Research Center, 1991. 2.3.1, 2.3.5

[CF58] Haskell Curry and Robert Feys. *Combinatory Logic*, volume 1. North Holland, 1958. 2.2

[CL90] Luca Cardelli and Xavier Leroy. Abstract types and the dot notation. In *Programming Concepts and Methods*, IFIP State of the Art Reports, pages 479–504. North Holland, March 1990. 2.3.1

[CL91] Luca Cardelli and Giuseppe Longo. A Semantic Basis for Quest. *Journal of Functional Programming*, 1(2):417—458, October 1991. 2.3.1

[Coq91] Thierry Coquand. An algorithm for testing conversion in Type Theory. In G. Huet and G. D. Plotkin, editors, *Logical Frameworks.* Cambridge University Press, 1991. 8.3.1

[Cou97a] Judicaël Courant. A Module Calculus for Pure Type Systems. In *TLCA'97*, LNCS, pages 112 — 128. Springer-Verlag, 1997. 3

[Cou97b] Judicaël Courant. An applicative module calculus. In *TAPSOFT'97*, LNCS. Springer-Verlag, April 1997. 3, 2.3.5, 11, 6.5.1

[Dam85] Luis Manuel Martins Damas. *Type Assignment in Programming Languages.* PhD thesis, Department of Computer Science, University of Edinburgh, April 1985. 3.2.3, 8.5

[DCH02] Derek Dreyer, Karl Crary, and Robert Harper. Moscow ML's higher-order modules are unsound, 17 September 2002. Message to the Types Forum `http://www.cis.upenn.edu/ bcpierce/types`. 10.1.4, 10.1.4

[deB72] N. G. deBruijn. Lambda calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Mathematics*, 34, 1972. 6.4.1

[Els99] Martin Elsman. *Program Modules, Separate Compilation, and Intermodule Optimisation.* PhD thesis, Dept. of Computer Science, University of Copenhagen, 1999. 10.1.10

[HH86] James G. Hook and Douglas J. Howe. Impredicative Strong Existential Equivalent to Type:Type. Technical Report TR 86-760, Department of Computer Science,Cornell University, June 1986. 2

[HL94] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *21st ACM Symp. Principles of Prog. Lang.*, 1994. 2.3.3, 3, 3, 2.3.5, 3.3.1, 6.4.1, 7.1, 9.2.1, 9.2.2

[HM93] Robert Harper and John C. Mitchell. On the type structure of Standard ML. In *ACM Trans. Prog. Lang. Syst.*, volume 15(2), pages 211–252, 1993. 2, 2.3.2, 2.3.2, 2.3.5, 3.3.1, 9.1, 9.2.1

[HMM86] Robert Harper, David MacQueen, and Robin Milner. Standard ML. Technical Report ECS-LFCS-86-2, Laboratory for Foundations of Computer Science, University of Edinburgh, May 1986. 2.3.1, 6.3.2

[HMM90] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. Technical Report ECS-LFCS-90-112, Department of Computer Science, University of Edinburgh, April 1990. 2.2.5, 2.3.2, 2.3.5, 3.3.1, 9.2.1

[How80] William Howard. The formulae-as-types notion of construction. In J. Hindley and J. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic*. Academic Press, 1980. 2.2

[HS97] Robert Harper and Chris Stone. An Interpretation of Standard ML in Type Theory. Technical Report CMU-CS-97-147, School of Computer Science,Carnegie Mellon University, June 1997. 2.3.3, 3, 3, 3.3.1, 7.1, 9.2.1, 9.2.2

[Hue75] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27—57, 1975. 5.6

[Jon96] Mark P. Jones. Using parameterized signatures to express modular structure. In *Proc. 23rd Symp. Principles of Prog. Lang.* ACM Press, 1996. 2.3.4

[KST97] Stefan Kahrs, Donald T. Sannella, and Andrzej Tarlecki. The Definition of Extended ML: A gentle introduction. *Theoretical Computer Science*, 173:445—484, 1997. 3.2.3

[Ler94] Xavier Leroy. Manifest types, modules, and separate compilation. In *Proc. 21st Symp. Principles of Prog. Lang.*, pages 109–122. ACM Press, 1994. 3, 3.3.1, 11, 6.1, 6.5.1, 8.2.1, 9.2.1, 9.2.2

[Ler95] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Proc. 22nd Symp. Principles of Prog. Lang.*, pages 142–153. ACM Press, 1995. 3, 3.3.1, 4.1.3, 4.1.1, 5.1, 5.2.3, 11, 6.5.1, 8.2.1, 9.2.1, 9.2.2, 9.2.3

[Ler96a] Xavier Leroy. A modular module system. Technical Report RR n° 2866, INRIA, Rocquencourt, April 1996. 3

[Ler96b] Xavier Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 6(5):1—32, September 1996. 3, 3.3.1, 5.1, 9.2.1, 9.2.2

[Ler97] Xavier Leroy. *The Objective Caml system, documentation and user's guide*, 1997. Available at `http://pauillac.inria.fr/ocaml`. 3, 1

[Lil97] Mark Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1997. 3, 3, 3.3.1, 5.1, 7.1, 9.2.1, 9.2.2

[Luo90] Zhaohui Luo. *An Extended Calculus of Constructions*. PhD thesis, Department of Computer Science, University of Edinburgh, June 1990. 2, 2

[Mac86] David MacQueen. Using dependent types to express modular structure. In *13th ACM Symp. on Principles of Prog. Lang.*, 1986. 2.3.1, 2.3.2, 2.3.5, 3.3.1, 5.2.2, 9.2.1

[MG93] Savi Maharaj and Elsa Gunter. Studying the ML Module System in HOL. *The Computer Journal*, 36(5), 1993. 2.3.4, 7.4.1, 9.3

[Mil78] Robin Milner. A theory of type polymorphism in programming languages. *Journal of Computer and System Sciences*, 17:348–375, 1978. 3.28, 3.2.3, 8.1, 8.1, 8.5

[Mil91] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 2/4:497 – 536, 1991. 5.6, 5.6

[Mil92] Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14:321 – 358, 1992. 8.3.1

[Mit96] John C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996. 2.2

[MP88] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988. 2.2.2, 2.3.1, 2.3.5, 7.2

[MP93] James McKinna and Robert Pollack. Pure Type Sytems formalized. In *Proc. Int'l Conf. on Typed Lambda Calculi and Applications, Utrecht*, pages 289–305, 1993. 4.3.2

[MT91] Robin Milner and Mads Tofte. *Commentary on Standard ML*. MIT Press, 1991. 2.1, 2.2, 6.3.2

[MT94] David MacQueen and Mads Tofte. A semantics for higher-order functors. In Donald Sannella, editor, *Programming Languages and Systems - ESOP '94*, volume 788 of *LNCS*. Springer Verlag, 1994. 2.3.3, 3, 2.3.4, 5.1

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990. 1.2, 2.1, 2.1.2, 2.1.1, 2.2, 2.3.2, 2.3.3, 3, 3, 3.1.2, 3.1.4, 3.3, 3.3.3, 10, 6.3.2, 9.3

[MTH96] Robin Milner, Mads Tofte, and Robert Harper. *The Revised Definition of Standard ML*. MIT Press, 1996. 1.2, 2.1, 2.1.2, 2.1.1, 2.2, 2.3.3, 3, 2.3.4, 3, 3.1.2, 3.1.4, 3.3, 3.3.3, 6.3.2, 9.3, 10

[NJ96] Jan Nicklish and Simon Peyton Jones. An exploration of modular programs. In *The Glasgow Workshop on Functional Programming*, 1996. 2.3.4

[Pau91] Larry Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991. 2.1

[Pot95] François Pottier. Implémentation d'un système de modules évolué en Caml-Light. Technical Report N° 2449, INRIA,Rocquencourt, January 1995. 3

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965. 8.1, 8.4, 8.3.1

[RRS00] Sergei Romanenko, Claudio V. Russo, and Peter Sestoft. Moscow ML Version 2.0, 2000. Available at `http://www.dina.kvl.dk/~sestoft/mosml`. (document), 1.2, 10

[Rus96] Claudio V. Russo. Standard ML Type Generativity as Existential Quantification. Technical Report ECS-LFCS-96-344, Laboratory for Foundations of Computer Science, University of Edinburgh, June 1996. 4

[Rus98a] Claudio V. Russo. An Implementation of First-Class Modules, March 1998. Available on request from the author. 1.3, 8.5, 9.3

[Rus98b] Claudio V. Russo. *Types For Modules*. PhD thesis, LFCS, University of Edinburgh, 1998. (document), 10

[Rus00a] Claudio V. Russo. First-Class Structures for Standard ML. In *European Symposium on Programming (ESOP)*, pages 336—350. Springer Verlag, 2000. 10.1

[Rus00b] Claudio V. Russo. First-Class Structures for Standard ML. *Nordic Journal of Computing*, 7(4):348—374, November 2000. 10.1

[Rus00c] Claudio V. Russo. The Definition of Non-Standard ML (Syntax and Static Semantics). Unpublished manuscript available on request, 2000. 10, 10.1.9

[Rus01] Claudio V. Russo. Recursive Structures for Standard ML. In *International Conference on Functional Programming (ICFP)*. ACM Press, 2001. 10.1, 10.1.7

[SH96] Chris Stone and Robert Harper. A Type-Theoretic Account of Standard ML 1996. Technical Report CMU-CS-96-136, School of Computer Science,Carnegie Mellon University, May 1996. 2.3.3, 3, 3, 3.3.1, 7.1, 9.2.1, 9.2.2

[Tho91] Simon Thompson. *Type Theory and Functional Programming*. Addison-Wesley, 1991. 2.2

[Tof88] Mads Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, Department of Computer Science, University of Edinburgh, May 1988. 2.3.4, 8.1, 8.1, 8.3.2

[Tof89] Mads Tofte. Four Lectures on Standard ML. Technical Report ECS-LFCS-89-73, Department of Computer Science, University of Edinburgh, March 1989. 2.1

[Tof92] Mads Tofte. Principal Signatures for Higher-Order Program Modules. In *Principles of Programming Languages*. ACM Press, January 1992. 2.3.4, 5.1

[Tof93] Mads Tofte. Principal Signatures for Higher-order Program Modules. *Journal of Functional Programming*, January 1993. 2.3.4, 5.1

[Wir88] Niklaus Wirth. *Programming in Modula-2*. Springer-Verlag, 4 edition, 1988. 3, 6.1

Index