Recursive Structures for Standard ML

Claudio V. Russo
Microsoft Research Ltd., St George House, 1 Guildhall Street, Cambridge CB2 3NH
crusso@microsoft.com

ABSTRACT

Standard ML is a statically typed programming language that is suited for the construction of both small and large programs. "Programming in the small" is captured by Standard ML's *Core* language. "Programming in the large" is captured by Standard ML's *Modules* language that 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. In Standard ML, modular programs must have a strictly hierarchical structure: the dependency between modules can never be cyclic. In particular, definitions of mutually recursive Core types and values, that arise frequently in practice, can never span module boundaries. This limitation compromises modular programming, forcing the programmer to merge conceptually (i.e. architecturally) distinct modules. We propose a practical and simple extension of the Modules language that caters for cyclic dependencies between both types and terms defined in separate modules. Our design leverages existing features of the language, supports separate compilation of mutually recursive modules and is easy to implement.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Modules,packages*

General Terms

LANGUAGES, THEORY

Keywords

recursive modules, datatypes, Standard ML, type theory

1. INTRODUCTION

Standard ML [12] (henceforth SML) is a high-level programming language that is suited for the construction of both small and large programs. SML's general-purpose *Core* language supports "programming in the small" with a rich range of types and computational constructs that includes mutually recursive datatypes and functions, control constructs, exceptions and references.

SML's special-purpose *Modules* language supports "programming in the large". Constructed on top of the Core, the Modules language allows sequential definitions of identifiers denoting Core language types and terms to be packaged together into possibly nested *structures*, whose components are accessed by the dot notation. Structures are *transparent*: by default, the *realisation* (i.e. implementation) of a type component within a structure is evident outside the structure. *Signatures* are used to specify the types of structures, by specifying their individual components. A type component may be specified *opaquely*, permitting a variety of realisations, or *transparently*, by equating it with a particular Core type ([10] uses the terminology *abstract* and *manifest* instead; we follow [8]). A structure *matches* a signature if it provides an implementation for all of the specified components, and, thanks to the subtyping relation called *enrichment*, possibly more. A signature may be used to *opaquely constrain* a matching structure. This existentially quantifies over the actual realisation of type components that have opaque specifications in the signature, effectively hiding their implementation. A *functor* definition defines a polymorphic function mapping structures to structures. A functor may be *applied* to any structure that realises a subtype of the formal argument's type, resulting in a concrete implementation of the functor body.

Despite the flexibility of the Modules type system, it does suffer from an awkward limitation. Unlike the definitions of the Core language, module bindings must have a strictly hierarchical structure: the dependency between modules can never be cyclic. In particular, although definable within the confines of a single module, definitions of mutually recursive types and values can never span module boundaries. While this does not affect the expressiveness of the Core language, the restriction does compromise modular programming. The programmer is typically left with two choices. She can either merge conceptually distinct modules into a single module, just to satisfy the type checker. Or she can resort to tricky encodings in the Core language to break the cycles between modules. The first approach obscures the architecture of the program; the second obscures its implementation. Neither solution is satisfactory.

In this article, we relax the hierarchical structure of Modules, allowing structures to be recursive. Our extension supports mutual recursion at the level of Core datatypes and (independently) values. Although different in detail, our proposal may be seen as adapting Crary, Harper and Puri's theoretical analysis of module recursion to Standard ML [2]. Our main contribution is to provide a practical type system for recursive modules in Standard ML that avoids some of the typing challenges of the formalism in [2], and exploits Standard ML's subtyping on modules to provide slightly more expressive and convenient constructs. Our extension preserves the existing features of Standard ML.

For presentation purposes, we formulate our extension, not for Standard ML, but for a representative toy language called Mini-SML. The static semantics of Mini-SML is based directly on that of Standard ML. Mini-SML includes the essential features of Standard ML Modules but, for brevity, only has a simple Core language of explicitly typed, monomorphic functions and non-parametric type and datatype definitions. [14] treats a more realistic Standard ML-like Core with implicitly typed, polymorphic functions and parameterised type definitions. Our proposal has been adapted to full Standard ML and is available in Moscow ML [17].

Section 2 introduces the syntax of Mini-SML. Sections 3 gives a motivating example to illustrate the limitations of acyclic Modules. Section 4 reviews the static semantics of Mini-SML. Section 5 defines our extension to recursive modules and its static and dynamic semantics. Section 6 recasts the first example so that its mutually dependent components may be separately compiled. Section 7 presents a real-world example that uses recursive modules to implement an advanced data structure. Section 8 assesses our contribution.

2. THE SYNTAX OF MINI-SML

Figure 1 defines the abstract syntax of Mini-SML. A *type path* tp is a projection t or sp.t of a (Core) type component from the context or a structure path. A *core type* u may be used to define a type identifier or to specify the type of a Core value. These are just the types of a simple functional language, extended with type paths. A *signature body* B is a sequential specification of a structure's components. A type component may be specified *transparently*, by equating it with a type, or *opaquely*, permitting a variety of *realisations* (ie. implementations). The implementation of a transparent component is fixed (up to the realisation of any opaque types that it mentions). An opaque datatype specification describes an arbitrary (recursive) datatype with finite set of *constructors* K. Each constructor $c \in K$ is specified to take n_c (≥ 0) arguments of type $u_{c,0} \cdots u_{c,n_c-1}$; n_c is the constructor's *arity*. In short, a datatype is a recursive, named sum of anonymous, possibly empty products The specification may be realised by any datatype with compatible constructors. Transparent datatype replication specifies a datatype that is equivalent to, and thus compatible with, the type tp (which must itself be bound to a datatype). Transparent types and datatype replication may be used to express *type sharing* constraints in the usual way. Value and structure components are specified by their type and signature. The specifications in a body are dependent in that subsequent specifications may refer to previous ones. A *signature expression* S encapsulates a body, or is a reference to a bound signature identifier. Informally, a structure *enriches* (has a subtype of) a completely transparent signature (ie. one containing no opaque type or datatype specifications) if it provides an implementation for all of its specified components, and possibly more. A structure *matches* a signature containing opaque type or datatype specifications if it enriches a complete realisation of that signature.

Core expressions e describe a simple functional language extended with the projection of a value identifier from a structure path. Constructor applications and case expressions are tagged with the name of the datatype (tp) that they introduce or eliminate. A constructor application takes the values of its n arguments and builds a tuple tagged with the constructor c, introducing a value of type tp. The typing rules ensure that the constructor is fully applied. A **case** expression evaluates e to a constructed value of type tp and chooses a continuation based on the tag of this value. K is a finite set of constructors used to index the set of alternative continuations. Each alternative binds n_c constructor arguments $x_{c,0} \cdots x_{c,n_c-1}$ in the continuation body e_c (typing will ensure that n_c is the arity of c in the datatype). A case expression need not be exhaustive, in which case evaluation aborts by raising the built-in exception **match**.

A *structure path* sp is a reference to a bound structure identifier, or the projection of one of its substructures. A *structure body* b is a dependent sequence of definitions: subsequent definitions may refer to previous ones. A type definition abbreviates a type. A datatype definition is like a datatype specification but generates a new (and thus distinct) type with the corresponding set of constructors. As in signatures, datatype replication declares a datatype that is equivalent to, and thus compatible with, the datatype tp. Datatype replication is used to copy a datatype into another scope whilst preserving compatibility with that type. Value, recursive function and structure definitions bind term identifiers to the values of expressions. A functor definition introduces a named function on structures: X is the functor's formal argument, S specifies the argument's type, and s is the functor's body that may refer to X. The functor may be applied to any argument that matches S. A signature definition abbreviates a signature. A *structure expression* s evaluates to a structure. It may be a path or an encapsulated structure body, whose type, value and structure definitions become the components of the structure. The application of a functor evaluates its body against the value of the actual argument. A transparent constraint (s : S) restricts the visibility of the structure's components to those specified in the signature, which the structure must match, and reveals the actual realisations of all type components in the signature (even those with opaque specifications). An opaque constraint (s :> S) is similar, but hides the actual realisation of type components with opaque specifications, introducing new abstract types.

3. MOTIVATING EXAMPLE

We can illustrate the limitations of the acyclic Modules language of Mini-SML (and Standard ML) by attempting to implement mutually recursive functions over mutually recursive datatypes. For more good examples, see [2].

Suppose we wish to define evaluation functions for two mutually recursive types of natural number and boolean expressions. Natural expressions include a conditional expression, `If`, that tests a boolean condition; boolean expressions include a predicate, `Null`, on naturals. For each sort of expression, the evaluation function `eval` reduces an expression to normal form.

Figure 2 is what we would like to write in Mini-SML, but

Meta-variables $\mathrm{t} \in \mathrm{TypId}$, $\mathrm{c} \in \mathrm{ConId}$, $\mathrm{x}, \mathrm{f} \in \mathrm{ValId}$, $\mathrm{X} \in \mathrm{StrId}$, $\mathrm{F} \in \mathrm{FunId}$ and $\mathrm{T} \in \mathrm{SigId}$ range over disjoint sets of type, constructor, value, structure, functor and signature identifiers.
The meta-variable K ranges over finite sets of constructor identifiers.
The meta-notation $\prod_{\mathrm{c} \in \mathrm{K}} \mathrm{p_c}$ and $\prod_{i<n} \mathrm{p}_i$ ranges over *finite sequences* of phrases p, indexed by constructor c or natural i.

Type Paths and Core Types

tp	$::=$	$\mathrm{t} \mid \mathrm{sp.t}$	type identifier, type projection
u	$::=$	$\mathrm{tp} \mid \mathrm{u} \to \mathrm{u}'$	type path, function type

Signature Bodies

B	$::=$	$\textbf{type}\ \mathrm{t} = \mathrm{u}; \mathrm{B}$	transparent type specification
	$\mid$	$\textbf{type}\ \mathrm{t}; \mathrm{B}$	opaque type specification
	$\mid$	$\textbf{datatype}\ \mathrm{t} = (\prod_{\mathrm{c} \in \mathrm{K}} \mathrm{c}\ \textbf{of}\ \prod_{i<n_\mathrm{c}} \mathrm{u}_{\mathrm{c},i})\ ; \mathrm{B}$	opaque datatype specification
	$\mid$	$\textbf{datatype}\ \mathrm{t} = \textbf{datatype}\ \mathrm{tp}\ ; \mathrm{B}$	transparent datatype replication
	$\mid$	$\textbf{val}\ \mathrm{x} : \mathrm{u}; \mathrm{B}$	value specification
	$\mid$	$\textbf{structure}\ \mathrm{X} : \mathrm{S}; \mathrm{B}$	structure specification
	$\mid$	ϵ_B	empty body

Signature Expressions

S	$::=$	$\textbf{sig}\ \mathrm{B}\ \textbf{end} \mid \mathrm{T}$	encapsulated body, signature identifier

Core Expressions

e	$::=$	x	value identifier
	$\mid$	$\lambda \mathrm{x} : \mathrm{u}.\mathrm{e}$	function
	$\mid$	$\mathrm{e}\ \mathrm{e}'$	application
	$\mid$	$\mathrm{sp.x}$	value projection
	$\mid$	$\mathrm{c}\ (\prod_{i<n} \mathrm{e}_i) : \mathrm{tp}$	constructor application
	$\mid$	$\textbf{case}\ \mathrm{e} : \mathrm{tp}\ \textbf{of}\ \prod_{\mathrm{c} \in \mathrm{K}} \mathrm{c} \prod_{i<n_\mathrm{c}} \mathrm{x}_{\mathrm{c},i} \Rightarrow \mathrm{e_c}$	constructor elimination

Structure Paths

sp	$::=$	$\mathrm{X} \mid \mathrm{sp.X}$	structure identifier, structure projection

Structure Bodies

b	$::=$	$\textbf{type}\ \mathrm{t} = \mathrm{u}; \mathrm{b}$	type definition
	$\mid$	$\textbf{datatype}\ \mathrm{t} = (\prod_{\mathrm{c} \in \mathrm{K}} \mathrm{c}\ \textbf{of}\ \prod_{i<n_\mathrm{c}} \mathrm{u}_{\mathrm{c},i})\ ; \mathrm{b}$	datatype definition
	$\mid$	$\textbf{datatype}\ \mathrm{t} = \textbf{datatype}\ \mathrm{tp}\ ; \mathrm{b}$	datatype replication
	$\mid$	$\textbf{val}\ \mathrm{x} = \mathrm{e}; \mathrm{b}$	value definition
	$\mid$	$\textbf{fun}\ \mathrm{f}(\mathrm{x} : \mathrm{u}) : \mathrm{u}' = \mathrm{e}; \mathrm{b}$	recursive function definition
	$\mid$	$\textbf{structure}\ \mathrm{X} = \mathrm{s};\mathrm{b}$	structure definition
	$\mid$	$\textbf{functor}\ \mathrm{F}\ (\mathrm{X} : \mathrm{S}) = \mathrm{s};\ \mathrm{b}$	functor definition
	$\mid$	$\textbf{signature}\ \mathrm{T} = \mathrm{S};\ \mathrm{b}$	signature definition
	$\mid$	ϵ_b	empty body

Structure Expressions

s	$::=$	sp	structure path
	$\mid$	$\textbf{struct}\ \mathrm{b}\ \textbf{end}$	structure body
	$\mid$	$\mathrm{F(s)}$	functor application
	$\mid$	$\mathrm{s} : \mathrm{S}$	transparent constraint
	$\mid$	$\mathrm{s} :> \mathrm{S}$	opaque constraint

(Mini-SML supports local functor and signature definitions so that structure bodies can play the role of Standard ML's separate top-level syntax. In SML, constructors can have a most one argument, with multiple arguments encoded as tuples — in Mini-SML, we support multiple arguments simply to avoid introducing separate syntax and rules for tuples. In SML, constructors are referenced by (long) value identifiers, with the constructor status (and associated datatype) recorded in the type of the identifier. To avoid formalizing this machinery, we simply treat constructors as uninterpreted tags, relying on explicit type annotations at constructor applications and case expressions to indicate membership of a particular datatype. Since we are only interested in cross-module recursion, datatype and function declarations are singly-recursive; support for mutual recursion is a by-product of our extension.)

Figure 1: Syntax of Mini-SML

```
structure Nat = struct
  datatype t = Zero | Succ of t
             | If of Bool.t[1] * t * t
  fun eval(n:t):t = case n : t of
      Zero => n
    | Succ m => Succ(eval m):t
    | If b t e => case Bool.eval[2] b:Bool.t[3] of
           True => eval t
         | False => eval e
end
structure Bool = struct
  datatype t = True | False | Null of Nat.t
  fun eval(b:t):t = case b : t of
     True => b
   | False => b
   | Null n => case Nat.eval n : Nat.t of
                 Zero => True:t
               | Succ m => False:t
end
```

Figure 2: mutually recursive structures

cannot. The code is rejected because of the three forward references to `Bool` (shown boxed) in the definition of `Nat`. Permuting the structure definitions does not help.

In Standard ML, the easiest way to define `Bool` and `Nat` is to use two global, simultaneous definitions (joined with **and**) of the datatypes and functions:

```
datatype  tNat = ... | If of tBool * tNat * tNat
     and tBool = ... | Null of tNat
fun evalNat(n:tNat):tNat = ... evalBool ...
and evalBool(n:tBool):tBool = ... evalNat ...
structure  Nat = struct datatype t = datatype tNat
                        val eval = evalNat
                 end
structure Bool = struct datatype t = datatype tBool
                        val eval = evalBool
                 end
```

In larger examples, this solution is unsatisfactory. It requires the programmer to mangle the names of the types and functions and define them in a scope that encompasses both modules, obscuring the program's architecture. The solution also impedes separate compilation, since most compilers do not allow simultaneous definitions to be split across compilation units. Another solution is to give a forward declaration, `ForwardNat`, whose individual components are parameterised by their forward references to `Bool`. `Bool` can then be defined in terms of `ForwardNat`. We subsequently define `Nat` so that its components are the fixed-points of applying `ForwardNat`'s components to the appropriate members of `Bool`. This solution is tedious, error-prone and inefficient, since the only way to represent `Bool.t` in `ForwardNat.eval` is as an abstract type, not a datatype.

It is instructive to analyse the forward references in Figure 2. The type reference $\boxed{\texttt{Bool.t}}^1$ merely refers to a type defined in `Bool` — the fact that `Bool.t` is a datatype with a set of constructors is irrelevant since we are neither trying to eliminate or introduce a value of the type. The term reference $\boxed{\texttt{Bool.eval}}^2$ refers to a value defined in `Bool`: the only information we need to typecheck this reference is its type. Finally, the type reference $\boxed{\texttt{Bool.t}}^3$ again refers to a type defined in `Bool`, but, unlike the first reference, the fact that `Bool.t` is actually a datatype is crucial in this context, since we are eliminating a value of the datatype and need to know its constructors and their types. If `Bool.t` was an ordinary type abbreviation or an abstract type, the case expression would fail to typecheck.

Suppose we now try to typecheck these forward references by throwing in some forward declarations. The first reference typechecks if we make a forward declaration that `Bool.t` is a type; the second typechecks if we make a forward declaration that `Bool.eval` has type `Bool.t -> Bool.t`; the third (and its enclosing case statement) typechecks if we make a forward declaration that `Bool.t` is the datatype `datatype t = True | False | Null of Nat.t`.

Inspired by the similar constructs in [2], we propose to extend the Modules language with two new constructs for expressing such forward declarations. The first construct is a new signature expression, **rec**(X:S_1)S_2, called a *recursive signature* (similar to the *recursively dependent signatures* of [2]). It constructs a signature from the body signature S_2, under the forward declaration of a structure X matching the signature S_1. Recursive signatures are useful for specifying mutually recursive types that span module boundaries within the signature.[1] Formally, we will require that the forward signature is enriched by (ie. a supertype of) the body, under some contractive (ie. non-circular) realisation of the type components in the forward specification. The realisation identifies datatypes in the body with any of their forward declarations as opaque types; it ties the recursive knot. The contractiveness condition forces each recursive type to be mediated by a datatype, which is consistent with ordinary SML. Allowing for enrichment lets us get away with only giving forward specifications for those components that are actually required in the body.

```
signature EVAL =
   rec(X: sig structure Bool: sig type t end end)
   sig structure Nat: sig
         datatype t = Zero | Succ of t
                    | If of X.Bool.t * t * t
       end
       structure Bool: sig
         datatype t = True | False | Null of Nat.t
         val eval: t -> t
       end
   end;
```

Figure 3: type recursion using a recursive signature

With a recursive signature, we can specify the mutually recursive datatypes declared in `Bool` and `Nat` (Figure 3). Notice that we only need a forward declaration of `X.Bool.t`, since the signature of `Bool` can make an ordinary (backward) reference to `Nat`. The specification of `Bool.eval` in the body, although superfluous here, will be used below.

The second construct, **rec**(X:S)s, is a new structure expression, called a *recursive structure*, that lets us construct

[1]The term *recursive signature* is perhaps a misnomer, since the forward declaration declares a structure, not a signature, and allows us to take the a fixed-point of the datatypes specified in the body, not a fixed-point of the body itself.

```
structure Eval = rec(X:EVAL)
  struct structure Nat = struct
           datatype t = datatype X.Nat.t
           fun eval(n:t):t = case n : t of
               Zero => n
             | Succ m => Succ(eval m):t
             | If b t e =>
                case X.Bool.eval b : X.Bool.t of
                  True => eval t
                | False => eval e
         end
         structure Bool = struct
           datatype t = datatype X.Bool.t
           fun eval(b:t):t = case b : t of
              True => b
            | False => b
            | Null n => case Nat.eval n : Nat.t of
                  Zero => True:t
                | Succ m => False:t
         end
  end
```

Figure 4: term recursion using a recursive structure

a cyclic value for s, given a forward declaration of this value as the structure X, and a partial specification of its type S. Recursive structures are used to define mutually recursive values whose definitions span module boundaries within s. Formally, we will require that the forward declaration of a recursive structure is enriched by the body. Allowing for enrichment lets us get away with only giving forward declarations for those components that are actually required in the body. Opaque datatypes and types in the forward declaration must be implemented in the body by datatype replication or an equivalent type abbreviation (respectively), and simply introduce new abstract types.

Now we can define the structures `Bool` and `Nat` as substructures of the recursive structure bound to `Eval` in Figure 3. The recursive structure is typechecked under the assumption that its body enriches the forward declaration of X (ie. that it matches the signature `EVAL`). The datatypes declared in `EVAL` are implemented by replicating them in the body. Since the function `X.Bool.eval` is declared in `EVAL` and the type `X.Bool.t` is declared as an appropriate datatype in `EVAL`, the boolean case expression in `Nat.eval` typechecks. Notice that the forward declaration only declares a subset of the body's components: we omitted `Nat.eval`, since no forward reference to it is required. `Eval.Nat.eval` will still be accessible, since the type of the body, not the forward declaration, determines the type of a recursive structure.

Operationally, a recursive structure is evaluated by evaluating its body under the initial assumption that X is undefined. If evaluation of the body attempts to evaluate X, execution aborts by raising the (new) exception $\perp$ (an alternative design is to enter a loop). If not, and evaluation of the body produces a value, we update the binding of X with this value and return the value as the value of the entire expression. This is similar to the treatment of recursive thunks in lazy languages, except that we evaluate recursion eagerly, to ensure that the order of any side-effects is deterministic. In our example, `Eval` is well-defined, since all (dynamic) references to X are delayed under abstractions.

Kinds, Type Variables, Variable Sets and Types

$$\begin{array}{rlr}
\kappa & \in \mathit{Kind} ::= * & \text{ranging over types} \\
 & \qquad\quad \mid \ \circ & \text{ranging over datatypes} \\
\alpha^\kappa & \in \mathit{Var}^\kappa \stackrel{\text{def}}{=} \{\alpha^\kappa, \beta^\kappa, \ldots\} & \text{kinded type variables} \\
\alpha & \in \mathit{Var} \stackrel{\text{def}}{=} \mathit{Var}^* \cup \mathit{Var}^\circ & \text{type variables} \\
P, Q & \in \mathit{VarSet} \stackrel{\text{def}}{=} \mathrm{Fin}(\mathit{Var}) & \text{type variable sets} \\
u & \in \mathit{Type} ::= \alpha & \text{type variable} \\
 & \qquad\quad \mid \ u \to u' & \text{function space}
\end{array}$$

Realisations

$$\varphi \in \mathit{Real} \stackrel{\text{def}}{=} \mathit{Var}^* \stackrel{\text{fin}}{\to} \mathit{Type} \cup \mathit{Var}^\circ \stackrel{\text{fin}}{\to} \mathit{Var}^\circ$$

Constructor Environments and Type Structures

$$\mathcal{K} \in \mathit{ConEnv} \stackrel{\text{def}}{=} \mathrm{ConId} \stackrel{\text{fin}}{\to} \textstyle\bigcup_{n \geq 0} \mathit{Type}^n$$

$$\Phi \in \mathit{TyStr} ::= u \mid (\alpha^\circ, \mathcal{K})$$

Structures, Signatures, and Existential Structures

$$\mathcal{S} \in \mathit{Str} \stackrel{\text{def}}{=} \left\{ \begin{array}{l|l} \mathcal{S}_\mathrm{t} \cup & \mathcal{S}_\mathrm{t} \in \mathrm{TypId} \stackrel{\text{fin}}{\to} \mathit{TyStr}, \\ \mathcal{S}_\mathrm{x} \cup & \mathcal{S}_\mathrm{x} \in \mathrm{ValId} \stackrel{\text{fin}}{\to} \mathit{Type}, \\ \mathcal{S}_\mathrm{X} & \mathcal{S}_\mathrm{X} \in \mathrm{StrId} \stackrel{\text{fin}}{\to} \mathit{Str} \end{array} \right\}$$

$$\mathcal{L} \in \mathit{Sig} ::= \Lambda P.\mathcal{S}$$

$$\mathcal{X} \in \mathit{ExStr} ::= \exists P.\mathcal{S}$$

Functors and Contexts

$$\mathcal{F} \in \mathit{Fun} ::= \forall P.\mathcal{S} \to \mathcal{X}$$

$$\mathcal{C} \in \mathit{Context} \stackrel{\text{def}}{=} \left\{ \begin{array}{l|l} \mathcal{C}_\mathrm{t} \cup & \mathcal{C}_\mathrm{t} \in \mathrm{TypId} \stackrel{\text{fin}}{\to} \mathit{TyStr}, \\ \mathcal{C}_\mathrm{T} \cup & \mathcal{C}_\mathrm{T} \in \mathrm{SigId} \stackrel{\text{fin}}{\to} \mathit{Sig}, \\ \mathcal{C}_\mathrm{x} \cup & \mathcal{C}_\mathrm{x} \in \mathrm{ValId} \stackrel{\text{fin}}{\to} \mathit{Type}, \\ \mathcal{C}_\mathrm{X} \cup & \mathcal{C}_\mathrm{X} \in \mathrm{StrId} \stackrel{\text{fin}}{\to} \mathit{Str}, \\ \mathcal{C}_\mathrm{F} & \mathcal{C}_\mathrm{F} \in \mathrm{FunId} \stackrel{\text{fin}}{\to} \mathit{Fun} \end{array} \right\}$$

Figure 5: semantic objects of Mini-SML

4. STATIC SEMANTICS OF MINI-SML

Before we can propose our extension, we need to review the static semantics, or typing judgements, of Mini-SML. Following [12], our static semantics distinguishes syntactic types of the language from their semantic counterparts, called *semantic objects*. The semantic objects, defined in Figure 5, play the role of types in the semantics. We let $\mathcal{O}$ range over all semantic objects.

Notation 1. For sets A and B, $\mathrm{Fin}(A)$ denotes the set of *finite subsets* of A, and $A \stackrel{\text{fin}}{\to} B$ denotes the set of *finite maps* from A to B. Let f and g be finite maps. $\mathcal{D}(f)$ denotes the *domain of definition* of f. The finite map $f + g$ has domain $\mathcal{D}(f) \cup \mathcal{D}(g)$ and values $(f+g)(a) \stackrel{\text{def}}{=}$ if $a \in \mathcal{D}(g)$ then $g(a)$ else $f(a)$. Finally, if $p \equiv \prod_{i<n} g_i$ is an n-tuple of finite maps we let $f \oplus p$ be the finite map defined inductively as follows: $(f \oplus p) \stackrel{\text{def}}{=} f$ if $n = 0$ and $(f \oplus p) \stackrel{\text{def}}{=} (f \oplus \prod_{i<m} g_i) + g_m$ if $n = m + 1$.

Type variables $\alpha \in \mathit{Var}$ are just *kinded* variables ranging over semantic types. Ordinary variables of kind $*$ range over arbitrary semantic types, but variables of kind $\circ$ are restricted to range only over other (datatype) variables. *Semantic types* $u \in \mathit{Type}$ are the semantic counterparts of syntactic Core types, and are used to record the denotations of type identifiers and the types of value identifiers. The symbols Λ, $\exists$ and $\forall$ bind finite sets, P, of type variables.

A *realisation* $\varphi \in \mathit{Real}$ maps type variables to semantic types and datatype variables to datatype variables. It defines a *substitution* on type variables (and a renaming on datatype variables) in the usual way. The operation of applying a realisation φ to an object $\mathcal{O}$ is written $\varphi(\mathcal{O})$. Realisations of datatype variables are restricted to renamings to ensure that the set of type structures (below) is closed under realisation.

Type structures $\Phi \in \mathit{TyStr}$ record the denotations of type components, and may either be a simple type u, arising from an ordinary type specification or definition, or a pair $(\alpha^\circ, \mathcal{K})$ of a datatype variable and its *constructor environment*, arising from a datatype specification or definition. $\mathcal{K} \in \mathit{ConEnv}$ is a finite map from constructors to tuples of argument types.

Semantic structures $\mathcal{S} \in \mathit{Str}$ are used as the types of structure identifiers and paths. A semantic structure maps type components to the type structures they denote, and value and structure components to the types they inhabit. For clarity, we define the extension functions $\mathrm{t} \triangleright \Phi, \mathcal{S} \stackrel{\mathrm{def}}{=} \{\mathrm{t} \mapsto \Phi\} + \mathcal{S}$, $\mathrm{x} : u, \mathcal{S} \stackrel{\mathrm{def}}{=} \{\mathrm{x} \mapsto u\} + \mathcal{S}$, and $\mathrm{X} : \mathcal{S}, \mathcal{S}' \stackrel{\mathrm{def}}{=} \{\mathrm{X} \mapsto \mathcal{S}\} + \mathcal{S}'$, and let $\epsilon_{\mathcal{S}}$ denote the empty structure $\emptyset$.

A *semantic signature* $\Lambda P.\mathcal{S}$ is a parameterised type: it describes the family of structures $\varphi(\mathcal{S})$, for φ a realisation of the parameters in P. Λ-bound type variables are introduced by datatype specifications and opaque specifications.

The *existential structure* $\exists P.\mathcal{S}$, on the other hand, is a quantified type: variables in P are existentially quantified in $\mathcal{S}$ and thus abstract. Existential structures describe the types of structure bodies and expression. Existentially quantified type variables are explicitly introduced by datatype definitions and opaque constraints s :> S, and implicitly eliminated at various points in the static semantics.

A *semantic functor* $\forall P.\mathcal{S} \rightarrow \mathcal{X}$ describes the type of a functor identifier: the universally quantified variables in P are bound simultaneously in the functor's domain, $\mathcal{S}$, and its range, $\mathcal{X}$. These variables capture the type components of the domain on which the functor behaves polymorphically; their possible occurrence in the range caters for the propagation of type identities from the functor's actual argument: functors are polymorphic functions on structures. The range $\mathcal{X}$ is the type of the functor body, that may introduce new (existential) types.

A *context* $\mathcal{C}$ maps type and signature identifiers to the type structures and signatures they denote, and maps value, structure and functor identifiers to the types they inhabit. For clarity, we define the extension functions $\mathcal{C}, \mathrm{t} \triangleright \Phi \stackrel{\mathrm{def}}{=} \mathcal{C} + \{\mathrm{t} \mapsto \Phi\}$, $\mathcal{C}, \mathrm{T} \triangleright \mathcal{L} \stackrel{\mathrm{def}}{=} \mathcal{C} + \{\mathrm{T} \mapsto \mathcal{L}\}$, $\mathcal{C}, \mathrm{x} : u \stackrel{\mathrm{def}}{=} \mathcal{C} + \{\mathrm{x} \mapsto u\}$, $\mathcal{C}, \mathrm{X} : \mathcal{S} \stackrel{\mathrm{def}}{=} \mathcal{C} + \{\mathrm{X} \mapsto \mathcal{S}\}$, and $\mathcal{C}, \mathrm{F} : \mathcal{F} \stackrel{\mathrm{def}}{=} \mathcal{C} + \{\mathrm{F} \mapsto \mathcal{F}\}$.

We let $\mathcal{V}(\mathcal{O})$ denote the set of type variables occurring *free* in $\mathcal{O}$, where the notions of free and bound variable are defined as usual. We also *identify* semantic objects that are equivalent up to α-conversion of bound type variables.

The operation of applying a realisation to a type (substitution) is extended to all semantic objects in the usual, capture-avoiding way.

Definition 1. **Enrichment Relation**

- Given two type structures Φ and Φ', Φ *enriches* Φ', written $\Phi \succeq \Phi'$, if and only if
  1. $\Phi = \Phi'$; or
  2. $\Phi \equiv (\alpha^\circ, \mathcal{K})$ and $\Phi' \equiv \alpha^\circ$.
- Given two structures $\mathcal{S}$ and $\mathcal{S}'$, $\mathcal{S}$ *enriches* $\mathcal{S}'$, written $\mathcal{S} \succeq \mathcal{S}'$, if and only if $\mathcal{D}(\mathcal{S}) \supseteq \mathcal{D}(\mathcal{S}')$ and
  1. for all $\mathrm{t} \in \mathcal{D}(\mathcal{S}')$, $\mathcal{S}(\mathrm{t}) \succeq \mathcal{S}'(\mathrm{t})$; and
  2. for all $\mathrm{x} \in \mathcal{D}(\mathcal{S}')$, $\mathcal{S}(\mathrm{x}) = \mathcal{S}'(\mathrm{x})$; and
  3. for all $\mathrm{X} \in \mathcal{D}(\mathcal{S}')$, $\mathcal{S}(\mathrm{X}) \succeq \mathcal{S}'(\mathrm{X})$.

Enrichment is a pre-order that defines a *subtyping* relation on semantic structures ($\mathcal{S} \succeq \mathcal{S}'$ means $\mathcal{S}$ is a subtype of $\mathcal{S}'$). The relation allows entire components to be forgotten in the supertype, but also allows a datatype to be equated with an ordinary type component, by hiding its constructors.

Definition 2. **Functor Instantiation**
A semantic functor $\forall P.\mathcal{S} \rightarrow \mathcal{X}$ *instantiates to* a *functor instance* $\mathcal{S}' \rightarrow \mathcal{X}'$, written $\forall P.\mathcal{S} \rightarrow \mathcal{X} > \mathcal{S}' \rightarrow \mathcal{X}'$, if and only if $\varphi(\mathcal{S}) = \mathcal{S}'$ and $\varphi(\mathcal{X}) = \mathcal{X}'$, for some realisation φ with $\mathcal{D}(\varphi) = P$.

Definition 3. **Signature Matching**
A semantic structure $\mathcal{S}'$ *matches* a signature $\Lambda P.\mathcal{S}$ if and only if $\mathcal{S}' \succeq \varphi(\mathcal{S})$ for some realisation φ with $\mathcal{D}(\varphi) = P$.

The static semantics of Mini-SML is defined by the judgements in Figures 6 and 7. Denotation judgements ($\mathcal{C} \vdash \mathrm{p} \triangleright \mathcal{O}$) relate type phrases to their denotations; classification judgements ($\mathcal{C} \vdash \mathrm{p} : \mathcal{O}$) relate term phrases to their semantic types. We deviate from the presentation in the Definition [12] by classifying structure expressions and bodies using existentially quantified semantic structures. The Definition classifies structure expressions using bare semantic structures, but, to capture generativity of type definitions, uses a state from which to generate new type variables . We prefer our presentation because it is stateless and thus more declarative ([6] uses a similar presentation). The procedural and declarative formalisations of the semantics are explained in detail, and proved equivalent, in [14, 15]. We only explain the additional rules concerning datatypes here.

Note that a constructor application or case expression is well-formed only if the explicit type path tp denotes a type structure with a constructor environment, ie. a datatype.

(**datatype** t $=$ $(\prod_{\mathrm{c} \in \mathrm{K}}$ c **of** $\prod_{i<n_{\mathrm{c}}} \mathrm{u}_{\mathrm{c},i})$; p): A datatype declaration, where p is a signature or structure body, is checked by building a constructor environment $\mathcal{K}$ from its constructor declarations, under the forward binding that t denotes a new (ordinary) type α°. The side condition $\alpha^\circ \notin \mathcal{V}(\mathcal{C})$ ensures that α° is not confused with any existing type in the context. The datatype t is added to the context with its constructor environment before classifying the remaining declarations in the body. A binding of the type structure for t is added to the resulting structure $\mathcal{S}$. The side condition on P prevents the capture of any free type variables in this type structure, by any of the parameters or existential types in P. In a signature, α° is a new type parameter; in a structure, it is a new existential type.

(**datatype** t $=$ **datatype** tp ; p) Datatype replication merely rebinds the existing type structure of the type tp to t, so that tp and t denote the same datatype in the body p.

$\boxed{\mathcal{C} \vdash \mathrm{tp} \rhd \Phi}$

$$\frac{\mathrm{t} \in \mathcal{D}(\mathcal{C})}{\mathcal{C} \vdash \mathrm{t} \rhd \mathcal{C}(\mathrm{t})} \qquad \frac{\mathcal{C} \vdash \mathrm{sp} : \mathcal{S} \quad \mathrm{t} \in \mathcal{D}(\mathcal{S})}{\mathcal{C} \vdash \mathrm{sp.t} \rhd \mathcal{S}(\mathrm{t})}$$

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

$$\frac{\mathcal{C} \vdash \mathrm{tp} \rhd u}{\mathcal{C} \vdash \mathrm{tp} \rhd u} \qquad \frac{\mathcal{C} \vdash \mathrm{tp} \rhd (\alpha^\circ, \mathcal{K})}{\mathcal{C} \vdash \mathrm{tp} \rhd \alpha^\circ}$$

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

$\boxed{\mathcal{C} \vdash \mathrm{B} \rhd \mathcal{L}}$

$$\frac{\begin{array}{cc} \mathcal{C} \vdash \mathrm{u} \rhd u & \mathcal{C}, \mathrm{t} \rhd u \vdash \mathrm{B} \rhd \Lambda P.\mathcal{S} \\ \mathrm{t} \notin \mathcal{D}(\mathcal{S}) & P \cap \mathcal{V}(u) = \emptyset \end{array}}{\mathcal{C} \vdash (\mathbf{type}\ \mathrm{t} = \mathrm{u}; \mathrm{B}) \rhd \Lambda P.(\mathrm{t} \rhd u, \mathcal{S})}$$

$$\frac{\alpha^* \notin \mathcal{V}(\mathcal{C}) \quad \mathcal{C}, \mathrm{t} \rhd \alpha^* \vdash \mathrm{B} \rhd \Lambda P.\mathcal{S} \quad \mathrm{t} \notin \mathcal{D}(\mathcal{S}) \quad \alpha^* \notin P}{\mathcal{C} \vdash (\mathbf{type}\ \mathrm{t}; \mathrm{B}) \rhd \Lambda\{\alpha^*\} \cup P.(\mathrm{t} \rhd \alpha^*, \mathcal{S})}$$

$$\frac{\begin{array}{l} \alpha^\circ \notin \mathcal{V}(\mathcal{C}) \\ \forall \mathrm{c} \in \mathrm{K}.\ \forall i < n_{\mathrm{c}}.\ \mathcal{C}, \mathrm{t} \rhd \alpha^\circ \vdash \mathrm{u}_{\mathrm{c},i} \rhd u_{\mathrm{c},i} \\ \mathcal{K} \equiv \{\mathrm{c} \mapsto \prod_{i<n_{\mathrm{c}}} u_{\mathrm{c},i} \mid \mathrm{c} \in \mathrm{K}\} \\ \mathcal{C}, \mathrm{t} \rhd (\alpha^\circ, \mathcal{K}) \vdash \mathrm{B} \rhd \Lambda P.\mathcal{S} \\ P \cap (\{\alpha^\circ\} \cup \mathcal{V}(\mathcal{K})) = \emptyset \\ \mathrm{t} \notin \mathcal{D}(\mathcal{S}) \end{array}}{\begin{array}{l} \mathcal{C} \vdash \quad (\mathbf{datatype}\ \mathrm{t}\ =\ (\prod_{\mathrm{c} \in \mathrm{K}}\ \mathrm{c}\ \mathbf{of}\ \prod_{i<n_{\mathrm{c}}} \mathrm{u}_{\mathrm{c},i})\ ; \mathrm{B}) \\ \qquad \rhd \Lambda\{\alpha^\circ\} \cup P.(\mathrm{t} \rhd (\alpha^\circ, \mathcal{K}), \mathcal{S}) \end{array}}$$

$$\frac{\begin{array}{ll} \mathcal{C} \vdash \mathrm{tp} \rhd (\alpha^\circ, \mathcal{K}) & \mathcal{C}, \mathrm{t} \rhd (\alpha^\circ, \mathcal{K}) \vdash \mathrm{B} \rhd \Lambda P.\mathcal{S} \\ P \cap (\{\alpha^\circ\} \cup \mathcal{V}(\mathcal{K})) = \emptyset & \mathrm{t} \notin \mathcal{D}(\mathcal{S}) \end{array}}{\begin{array}{l} \mathcal{C} \vdash \quad (\mathbf{datatype}\ \mathrm{t}\ =\ \mathbf{datatype}\ \mathrm{tp}\ ; \mathrm{B}) \\ \qquad \rhd \Lambda P.(\mathrm{t} \rhd (\alpha^\circ, \mathcal{K}), \mathcal{S}) \end{array}}$$

$$\frac{\begin{array}{cc} \mathcal{C} \vdash \mathrm{u} \rhd u & \mathcal{C}, \mathrm{x} : u \vdash \mathrm{B} \rhd \Lambda P.\mathcal{S} \\ \mathrm{x} \notin \mathcal{D}(\mathcal{S}) & P \cap \mathcal{V}(u) = \emptyset \end{array}}{\mathcal{C} \vdash (\mathbf{val}\ \mathrm{x} : \mathrm{u}; \mathrm{B}) \rhd \Lambda P.(\mathrm{x} : u, \mathcal{S})}$$

$$\frac{\begin{array}{ll} \mathcal{C} \vdash \mathrm{S} \rhd \Lambda P.\mathcal{S} & P \cap \mathcal{V}(\mathcal{C}) = \emptyset \\ \mathcal{C}, \mathrm{X} : \mathcal{S} \vdash \mathrm{B} \rhd \Lambda Q.\mathcal{S}' & \\ \mathrm{X} \notin \mathcal{D}(\mathcal{S}') & Q \cap (P \cup \mathcal{V}(\mathcal{S})) = \emptyset \end{array}}{\mathcal{C} \vdash (\mathbf{structure}\ \mathrm{X} : \mathrm{S}; \mathrm{B}) \rhd \Lambda P \cup Q.(\mathrm{X} : \mathcal{S}, \mathcal{S}')}$$

$$\frac{}{\mathcal{C} \vdash \epsilon_{\mathrm{B}} \rhd \Lambda\emptyset.\epsilon_{\mathcal{S}}}$$

$\boxed{\mathcal{C} \vdash \mathrm{S} \rhd \mathcal{L}}$

$$\frac{\mathcal{C} \vdash \mathrm{B} \rhd \Lambda P.\mathcal{S}}{\mathcal{C} \vdash \mathbf{sig}\ \mathrm{B}\ \mathbf{end} \rhd \Lambda P.\mathcal{S}} \qquad \frac{\mathrm{T} \in \mathcal{D}(\mathcal{C})}{\mathcal{C} \vdash \mathrm{T} \rhd \mathcal{C}(\mathrm{T})}$$

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

$$\frac{\mathrm{x} \in \mathcal{D}(\mathcal{C})}{\mathcal{C} \vdash \mathrm{x} : \mathcal{C}(\mathrm{x})}$$

$$\frac{\mathcal{C} \vdash \mathrm{u} \rhd u \quad \mathcal{C}, \mathrm{x} : u \vdash \mathrm{e} : u'}{\mathcal{C} \vdash \lambda \mathrm{x} : \mathrm{u.e} : u \to u'} \qquad \frac{\mathcal{C} \vdash \mathrm{e} : u' \to u \quad \mathcal{C} \vdash \mathrm{e}' : u'}{\mathcal{C} \vdash \mathrm{e}\ \mathrm{e}' : u}$$

$$\frac{\begin{array}{ll} \mathcal{C} \vdash \mathrm{tp} \rhd (\alpha^\circ, \mathcal{K}) & \mathrm{c} \in \mathcal{D}(\mathcal{K}) \\ \mathcal{K}(\mathrm{c}) = \Pi_{i<n} u_i & \forall i < n.\mathcal{C} \vdash \mathrm{e}_i : u_i \end{array}}{\mathcal{C} \vdash (\mathrm{c}\ (\prod_{i<n} \mathrm{e}_i) : \mathrm{tp}) : \alpha^\circ}$$

$$\frac{\begin{array}{ll} \mathcal{C} \vdash \mathrm{e} : \alpha^\circ & \mathcal{C} \vdash \mathrm{tp} \rhd (\alpha^\circ, \mathcal{K}) \\ \forall \mathrm{c} \in \mathrm{K}. & \mathrm{c} \in \mathcal{D}(\mathcal{K}) \wedge \mathcal{K}(\mathrm{c}) = \Pi_{i<n_{\mathrm{c}}} u_{\mathrm{c},i} \wedge \\ & \mathcal{C} \oplus \Pi_{i<n_{\mathrm{c}}}\{\mathrm{x}_{\mathrm{c},i} \mapsto u_{\mathrm{c},i}\} \vdash \mathrm{e}_{\mathrm{c}} : u' \end{array}}{\mathcal{C} \vdash (\mathbf{case}\ \mathrm{e} : \mathrm{tp}\ \mathbf{of}\ \prod_{\mathrm{c} \in \mathrm{K}} \mathrm{c} \prod_{i<n_{\mathrm{c}}} \mathrm{x}_{\mathrm{c},i}\ \Rightarrow\ \mathrm{e}_{\mathrm{c}}) : u'}$$

$$\frac{\mathcal{C} \vdash \mathrm{sp} : \mathcal{S} \quad \mathrm{x} \in \mathcal{D}(\mathcal{S})}{\mathcal{C} \vdash \mathrm{sp.x} : \mathcal{S}(\mathrm{x})}$$

Figure 6: Denotation and Core classification judgements

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

$$\frac{\mathrm{X} \in \mathcal{D}(\mathcal{C})}{\mathcal{C} \vdash \mathrm{X} : \mathcal{C}(\mathrm{X})}$$

$$\frac{\mathcal{C} \vdash \mathrm{sp} : \mathcal{S} \quad \mathrm{X} \in \mathcal{D}(\mathcal{S})}{\mathcal{C} \vdash \mathrm{sp.X} : \mathcal{S}(\mathrm{X})}$$

$\boxed{\mathcal{C} \vdash \mathrm{s} : \mathcal{X}}$

$$\frac{\mathcal{C} \vdash \mathrm{sp} : \mathcal{S}}{\mathcal{C} \vdash \mathrm{sp} : \exists\emptyset.\mathcal{S}}$$

$$\frac{\mathcal{C} \vdash \mathrm{b} : \exists P.\mathcal{S}}{\mathcal{C} \vdash \mathbf{struct}\ \mathrm{b}\ \mathbf{end} : \exists P.\mathcal{S}}$$

$$\frac{\begin{array}{lll} \mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S} & P \cap \mathcal{V}(\mathcal{C}(\mathrm{F})) = \emptyset & \\ \mathcal{C}(\mathrm{F}) > \mathcal{S}' \to \exists Q.\mathcal{S}'' & \mathcal{S} \succeq \mathcal{S}' & Q \cap P = \emptyset \end{array}}{\mathcal{C} \vdash \mathrm{F}(\mathrm{s}) : \exists P \cup Q.\mathcal{S}''}$$

$$\frac{\begin{array}{lll} \mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S} & \mathcal{C} \vdash \mathrm{S} \rhd \Lambda Q.\mathcal{S}' & \\ P \cap \mathcal{V}(\Lambda Q.\mathcal{S}') = \emptyset & \mathcal{S} \succeq \varphi(\mathcal{S}') & \mathcal{D}(\varphi) = Q \end{array}}{\mathcal{C} \vdash (\mathrm{s} : \mathrm{S}) : \exists P.\varphi(\mathcal{S}')}$$

$$\frac{\begin{array}{lll} \mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S} & \mathcal{C} \vdash \mathrm{S} \rhd \Lambda Q.\mathcal{S}' & \\ P \cap \mathcal{V}(\Lambda Q.\mathcal{S}') = \emptyset & \mathcal{S} \succeq \varphi(\mathcal{S}') & \mathcal{D}(\varphi) = Q \end{array}}{\mathcal{C} \vdash (\mathrm{s} :> \mathrm{S}) : \exists Q.\mathcal{S}'}$$

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

$$\frac{\mathcal{C} \vdash \mathrm{u} \rhd u \quad \mathcal{C}, \mathrm{t} \rhd u \vdash \mathrm{b} : \exists P.\mathcal{S} \quad P \cap \mathcal{V}(u) = \emptyset}{\mathcal{C} \vdash (\mathbf{type}\ \mathrm{t} = \mathrm{u}; \mathrm{b}) : \exists P.(\mathrm{t} \rhd u, \mathcal{S})}$$

$$\frac{\begin{array}{l} \alpha^\circ \notin \mathcal{V}(\mathcal{C}) \\ \forall \mathrm{c} \in \mathrm{K}.\ \forall i < n_{\mathrm{c}}.\ \mathcal{C}, \mathrm{t} \rhd \alpha^\circ \vdash \mathrm{u}_{\mathrm{c},i} \rhd u_{\mathrm{c},i} \\ \mathcal{K} \equiv \{\mathrm{c} \mapsto \prod_{i<n_{\mathrm{c}}} u_{\mathrm{c},i} \mid \mathrm{c} \in \mathrm{K}\} \\ \mathcal{C}, \mathrm{t} \rhd (\alpha^\circ, \mathcal{K}) \vdash \mathrm{b} : \exists P.\mathcal{S} \\ P \cap (\{\alpha^\circ\} \cup \mathcal{V}(\mathcal{K})) = \emptyset \end{array}}{\begin{array}{l} \mathcal{C} \vdash \quad (\mathbf{datatype}\ \mathrm{t}\ =\ (\prod_{\mathrm{c} \in \mathrm{K}}\ \mathrm{c}\ \mathbf{of}\ \prod_{i<n_{\mathrm{c}}} \mathrm{u}_{\mathrm{c},i})\ ; \mathrm{b}) \\ \qquad : \exists\{\alpha^\circ\} \cup P.(\mathrm{t} \rhd (\alpha^\circ, \mathcal{K}), \mathcal{S}) \end{array}}$$

$$\frac{\begin{array}{l} \mathcal{C} \vdash \mathrm{tp} \rhd (\alpha^\circ, \mathcal{K}) \quad \mathcal{C}, \mathrm{t} \rhd (\alpha^\circ, \mathcal{K}) \vdash \mathrm{b} : \exists P.\mathcal{S} \\ P \cap (\{\alpha^\circ\} \cup \mathcal{V}(\mathcal{K})) = \emptyset \end{array}}{\begin{array}{l} \mathcal{C} \vdash \quad (\mathbf{datatype}\ \mathrm{t}\ =\ \mathbf{datatype}\ \mathrm{tp}\ ; \mathrm{b}) \\ \qquad : \exists P.(\mathrm{t} \rhd (\alpha^\circ, \mathcal{K}), \mathcal{S}) \end{array}}$$

$$\frac{\mathcal{C} \vdash \mathrm{e} : u \quad \mathcal{C}, \mathrm{x} : u \vdash \mathrm{b} : \exists P.\mathcal{S} \quad P \cap \mathcal{V}(u) = \emptyset}{\mathcal{C} \vdash (\mathbf{val}\ \mathrm{x} = \mathrm{e}; \mathrm{b}) : \exists P.(\mathrm{x} : u, \mathcal{S})}$$

$$\frac{\begin{array}{ll} \mathcal{C} \vdash \mathrm{u} \rhd u & \mathcal{C} \vdash \mathrm{u}' \rhd u' \\ \mathcal{C}, \mathrm{f} : u \to u', \mathrm{x} : u \vdash \mathrm{e} : u' & \\ \mathcal{C}, \mathrm{f} : u \to u' \vdash \mathrm{b} : \exists P.\mathcal{S} & P \cap \mathcal{V}(u \to u') = \emptyset \end{array}}{\mathcal{C} \vdash (\mathbf{fun}\ \mathrm{f}(\mathrm{x} : \mathrm{u}) : \mathrm{u}' = \mathrm{e}; \mathrm{b}) : \exists P.(\mathrm{f} : u \to u', \mathcal{S})}$$

$$\frac{\begin{array}{ll} \mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S} & P \cap \mathcal{V}(\mathcal{C}) = \emptyset \\ \mathcal{C}, \mathrm{X} : \mathcal{S} \vdash \mathrm{b} : \exists Q.\mathcal{S}' & Q \cap (P \cup \mathcal{V}(\mathcal{S})) = \emptyset \end{array}}{\mathcal{C} \vdash (\mathbf{structure}\ \mathrm{X}\ =\ \mathrm{s;b}) : \exists P \cup Q.(\mathrm{X} : \mathcal{S}, \mathcal{S}')}$$

$$\frac{\begin{array}{ll} \mathcal{C} \vdash \mathrm{S} \rhd \Lambda P.\mathcal{S} & P \cap \mathcal{V}(\mathcal{C}) = \emptyset \\ \mathcal{C}, \mathrm{X} : \mathcal{S} \vdash \mathrm{s} : \mathcal{X} & \mathcal{C}, \mathrm{F} : \forall P.\mathcal{S} \to \mathcal{X} \vdash \mathrm{b} : \exists Q.\mathcal{S} \end{array}}{\mathcal{C} \vdash (\mathbf{functor}\ \mathrm{F}\ (\mathrm{X} : \mathrm{S})\ =\ \mathrm{s};\ \mathrm{b}) : \exists Q.\mathcal{S}}$$

$$\frac{\mathcal{C} \vdash \mathrm{S} \rhd \mathcal{L} \quad \mathcal{C}, \mathrm{T} \rhd \mathcal{L} \vdash \mathrm{b} : \exists P.\mathcal{S}}{\mathcal{C} \vdash (\mathbf{signature}\ \mathrm{T}\ =\ \mathrm{S};\ \mathrm{b}) : \exists P.\mathcal{S}} \qquad \frac{}{\mathcal{C} \vdash \epsilon_{\mathrm{b}} : \exists\emptyset.\epsilon_{\mathcal{S}}}$$

Figure 7: Modules classification judgements

5. THE EXTENSION

Formally, our extension requires two new syntactic constructs, both additions to the Modules language:

$$
\begin{array}{llcll}
\mathrm{S} & ::= & \ldots & | & \mathbf{rec}(\mathrm{X{:}S})\mathrm{S}' & \text{recursive signature} \\
\mathrm{s} & ::= & \ldots & | & \mathbf{rec}(\mathrm{X{:}S})\mathrm{s} & \text{recursive structure}
\end{array}
$$

Two new rules extend the Modules judgements $\mathcal{C} \vdash \mathrm{S} \triangleright \mathcal{L}$ and $\mathcal{C} \vdash \mathrm{s} : \mathcal{X}$ (the semantic objects are unchanged):

$$
\frac{\begin{array}{ll}\mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{S} & P \cap \mathcal{V}(\mathcal{C}) = \emptyset \\ \mathcal{C}, \mathrm{X} : \mathcal{S} \vdash \mathrm{S}' \triangleright \Lambda Q.\mathcal{S}' & Q \cap (P \cup \mathcal{V}(\mathcal{S})) = \emptyset \\ \varphi(\mathcal{S}') \succeq \varphi(\mathcal{S}) \quad \mathcal{D}(\varphi) = P & P \cap \bigcup_{\alpha \in P} \mathcal{V}(\varphi(\alpha)) = \emptyset \end{array}}{\mathcal{C} \vdash (\mathbf{rec}(\mathrm{X{:}S})\mathrm{S}') \triangleright \Lambda Q.\varphi(\mathcal{S}')} \quad (1)
$$

Rule 1 relates a recursive signature expression to its denotation, a semantic signature. The parameters P of the semantic signature $\Lambda P.\mathcal{S}$ stem from opaque type and datatype specifications in the forward declaration's syntactic signature S. The denotation of the signature body, S$'$, is determined in the current context, extended with the assumption that X has type $\mathcal{S}$. The side-condition $P \cap \mathcal{V}(\mathcal{C}) = \emptyset$ prevents the capture of free variables in $\mathcal{C}$ by the bound variables in P and ensures that these variables are treated as parameters for the classification of S$'$. The forward parameters P, which may occur in $\mathcal{S}$ and thus in $\Lambda Q.\mathcal{S}'$, must be realised in a way that ensures the body enriches the forward specification. The realisation φ identifies type variables stemming from opaque and datatype specifications in the forward signature with types specified in the body. Note that the variables in P may occur in $\mathcal{S}'$, so we apply the realisation to *both* structures when checking enrichment; the simultaneous realisation ties the recursive knot. The first side condition on the domain of φ ensures that all forward specifications are realised. The second side condition on the range of φ ensures that the realisation is not circular, hence contractive and unique. The parameters Q of the body determine the parameters of the entire signature $\Lambda Q.\varphi(\mathcal{S}')$.

$$
\frac{\begin{array}{ll}\mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{S} & P \cap \mathcal{V}(\mathcal{C}) = \emptyset \\ \mathcal{C}, \mathrm{X} : \mathcal{S} \vdash \mathrm{s} : \exists Q.\mathcal{S}' & \\ Q \cap (P \cup \mathcal{V}(\mathcal{S})) = \emptyset & \mathcal{S}' \succeq \mathcal{S} \end{array}}{\mathcal{C} \vdash (\mathbf{rec}(\mathrm{X{:}S})\mathrm{s}) : \exists P \cup Q.\mathcal{S}'} \quad (2)
$$

Rule 2 relates a recursive structure expression to its type. If the forward declaration's signature S denotes $\Lambda P.\mathcal{S}$, the body s of the structure expression is classified in the extended context $\mathcal{C}, \mathrm{X} : \mathcal{S}$. The side-condition $P \cap \mathcal{V}(\mathcal{C}) = \emptyset$ prevents the capture of free variables in $\mathcal{C}$ by the bound variables in P and ensures that these variables from the forward declaration are treated as new types for the classification of s. During this classification, the constructors of any datatypes and the types of any values specified in S are known. Since S is intended as a forward declaration of the types and values in s, the rule requires that the type of s, $\exists Q.\mathcal{S}'$, enriches the forward declaration. In particular, this checks that any forward declared values that might be referenced in s are actually defined with the correct type in s (when the value of s is defined). Note that s may itself declare some new types Q. Before checking that the type of the body enriches the forward declaration, we eliminate the existential quantification over Q, ensuring that these hypothetical types cannot capture any types in the forward declaration. The new types returned by the phrase are the union of the new types in the forward declaration and the new types of the body.

Although not fully illustrated by our examples, Rule 1 allows a mixture of opaque and transparent type specifications in both the forward declaration and the body. Opaque type components in the forward declaration must be specified in the body as either a non-circular transparent type, an (inherently non-circular) opaque type, or a potentially circular datatype; transparent type components must simply be implemented by equivalent types in the body (modulo realisation by φ). (The side condition on the range of φ rules out the declaration of equi-recursive types - we only support recursion through named, iso-recursive types. At higher kinds (type constructors), this restriction appears to avoid the typing difficulties associated with equi-recursive types, whose equivalence at higher kinds is not known to be decidable [2].) Unlike Rule 1, Rule 2 merely requires that the type of the body enriches the forward declaration, without allowing the further realisation of any opaque types in the forward signature. Omitting the realisation step is a design decision but is motivated by the principle that type equivalences valid outside the body should also be valid inside it: in particular, a forward reference to a type component should be compatible with a backward reference to its corresponding declaration in the body.

Given a semantic signature $\Lambda P.\mathcal{S}$, and a semantic structure $\mathcal{S}'$, the algorithm for matching the structure against the signature is a straightforward adaptation of the folklore two-pass algorithm (a simpler, one-pass variant, suitable in the absence of recursive types, is described and proved correct in [14]). The first-pass of the algorithm is used to construct a candidate realisation φ of the variables in P. The algorithm traverses the structure of $\mathcal{S}$ keeping $\mathcal{S}'$ fixed. Each type variable α in P (and only those in P) is realised incrementally (by the corresponding type component in $\mathcal{S}'$), as we encounter its first occurrence in a type structure of the form α or $(\alpha, \mathcal{K})$ in $\mathcal{S}$ (corresponding to the opaque or datatype specification that introduced α). The incremental realisation is applied to the current realisation of $\mathcal{S}$ before resuming the traversal. After the construction of φ, the folklore algorithm conducts a second traversal of $\varphi(\mathcal{S})$, to check that the original $\mathcal{S}'$ enriches $\varphi(\mathcal{S})$. Our algorithm, however, must also be able to check the enrichment condition in Rule 1, namely $\varphi(\mathcal{S}') \succeq \varphi(\mathcal{S})$ for some φ with $\mathcal{D}(\varphi){=}P$ where φ is not circular ($P \cap \bigcup_{\alpha \in P} \mathcal{V}(\varphi(\alpha)) = \emptyset$). Observe that enrichment must hold under a *simultaneous* realisation of not only $\mathcal{S}$, but also $\mathcal{S}'$, which is different from the ordinary signature matching problem where $\mathcal{S}'$ is assumed not to mention any of the variables in P. Our solution is to use a modified signature matching algorithm, that, during the first traversal of $\mathcal{S}$, performs an occur check at each realisation step, preventing variables in P from appearing in the range of φ. In addition, before resuming the traversal, the incremental realisation is applied not only to the current realisation of $\mathcal{S}$, but also of $\mathcal{S}'$. The second pass of the algorithm simply checks that $\varphi(\mathcal{S}') \succeq \varphi(\mathcal{S})$. Note that the algorithm is very similar to unification. The algorithm and its correctness proof will appear in a future technical report. It easily scales to cope with Standard ML's type constructors (parameterised types).

We can present the *dynamic semantics* of recursive structures as a simple extension of a sketched call-by-value semantics for Mini-SML. The Mini-SML semantics is intended to model the semantics of Standard ML. Our presentation follows the style of the Definition [12]. The dynamic objects used during evaluation consist of:

- a set of *exceptions* ex $\in$ Exn (distinct from values) that includes the existing exception **match** raised by the attempted evaluation of a missing **case** alternative, and a new exception $\bot$, raised by a premature reference to an undefined recursive structure identifier.
- a set of *core values* v $\in$ CorVal that includes recursive function closures and applications of n-ary constructors to values (whose form we shall leave unspecified).
- a set of *structure values*, mapping value and structure identifiers to values:

$$\mathrm{V} \in \mathrm{StrVal} \overset{\text{def}}{=} \left\{ \begin{array}{l|l} \mathrm{V_x} \cup & \mathrm{V_x} \in \mathrm{ValId} \overset{\text{fin}}{\to} \mathrm{CorVal} \\ \mathrm{V_X} & \mathrm{V_X} \in \mathrm{StrId} \overset{\text{fin}}{\to} \mathrm{StrVal} \end{array} \right\}$$

- a set of *core results* r $\in$ CorRes ::= v | ex to capture the values returned or exceptions raised by Core evaluation.
- a set of *structure results* R $\in$ StrRes ::= V | ex to capture the values returned or exceptions raised by Module evaluation.
- a set of *functor closures* $<\mathrm{X}, \mathcal{E}, \mathrm{s}> \in$ FunVal.
- a infinite set of heap locations $l \in$ Loc, distinct from StrVal.
- a set of finite *heaps* mapping locations to structure values or the distinguished undefined element $\bot$:

$$\mathcal{H} \in \mathrm{Heap} \overset{\text{def}}{=} \mathrm{Loc} \overset{\text{fin}}{\to} \mathrm{StrVal} \cup \{\bot\}$$

- a set of environments $\mathcal{E} \in$ Env mapping value identifiers to core values, structure identifiers to structure values or locations and functor identifiers to closures:

$$\mathcal{E} \in \mathrm{Env} \overset{\text{def}}{=} \left\{ \begin{array}{l|l} \mathcal{E}_\mathrm{x} \cup & \mathcal{E}_\mathrm{x} \in \mathrm{ValId} \overset{\text{fin}}{\to} \mathrm{CorVal} \\ \mathcal{E}_\mathrm{X} \ \cup & \mathcal{E}_\mathrm{X} \in \mathrm{StrId} \overset{\text{fin}}{\to} \mathrm{StrVal} \cup \mathrm{Loc} \\ \mathcal{E}_\mathrm{F} & \mathcal{E}_\mathrm{F} \in \mathrm{FunId} \overset{\text{fin}}{\to} \mathrm{FunVal} \end{array} \right\}$$

The dynamic semantics is defined by environment based evaluation judgements (Figure 8). Each judgement form relates an environment, initial heap and expression to a pair, consisting of a result (a value or an exception) and a possibly updated heap. The omitted judgements $\mathcal{E}, \mathcal{H} \vdash \mathrm{e} \downarrow \mathrm{r}, \mathcal{H}'$ $\mathcal{E}, \mathcal{H} \vdash \mathrm{b} \downarrow \mathrm{R}, \mathcal{H}'$ for Core expressions and structure bodies are a straightforward adaptation of the existing Standard ML rules, modified to propagate a heap in the same way that Standard ML's evaluation rules already propagate a store (in an implementation, the store can simply be re-used for the heap). Recursive structure expressions are evaluated by choosing a new location in the heap, initialized to $\bot$. If evaluation of the recursive structure's body yields a value, we update the heap and return that value (Rule (6)). If it yields an exception, perhaps because of a premature evaluation of X, we simply propagate the exception (Rule(7)). Although we have omitted many of the rules, as in Standard ML, structure declarations and functor applications always bind structure identifiers to values, not locations. Evaluating a structure identifier that refers to a functor argument or structure declaration returns the value bound to that identifier in the environment (Rule (1)). If the identifier refers to a recursively bound identifier, which is necessarily bound to a location, we dereference that location in the heap, and return the stored value (Rule (2)), or raise the exception $\bot$ if the location contains $\bot$ (Rule (3)). Evaluating a projection either projects a value from the path's value, or propagates an exception (Rules (5) and (6)). In an implementation, we can distinguish references to recursive and non-recursive bindings statically, so there is no need for a run-time test to distinguish applications of Rules (2) or (3) from Rule (1). This is important because it ensures that the compilation of references to non-recursive bindings, that occur in ordinary Standard ML programs, is not penalised by our extension. Since we evaluate the bodies of recursive structures eagerly, and only once, the execution order of their side-effects (if any) is completely deterministic: this means that it is straightforward to extend Mini-SML with the impure features of ordinary SML (references, exceptions and I/O) that rely on a fixed evaluation order. Note that functor application remains call-by-value (Rules (8) and (9)).

$$\boxed{\mathcal{E}, \mathcal{H} \vdash \mathrm{sp} \downarrow \mathrm{R}, \mathcal{H}'} \qquad \frac{\mathrm{X} \in \mathcal{D}(\mathcal{E}) \quad \mathcal{E}(\mathrm{X}) = \mathrm{V}}{\mathcal{E}, \mathcal{H} \vdash \mathrm{X} \downarrow \mathrm{V}, \mathcal{H}} \ (1)$$

$$\frac{\mathrm{X} \in \mathcal{D}(\mathcal{E}) \quad \mathcal{E}(\mathrm{X}) = l \quad l \in \mathcal{D}(\mathcal{H}) \quad \mathcal{H}(l) = \mathrm{V}}{\mathcal{E}, \mathcal{H} \vdash \mathrm{X} \downarrow \mathrm{V}, \mathcal{H}} \ (2)$$

$$\frac{\mathrm{X} \in \mathcal{D}(\mathcal{E}) \quad \mathcal{E}(\mathrm{X}) = l \quad l \in \mathcal{D}(\mathcal{H}) \quad \mathcal{H}(l) = \bot}{\mathcal{E}, \mathcal{H} \vdash \mathrm{X} \downarrow \bot, \mathcal{H}} \ (3)$$

$$\frac{\mathcal{E}, \mathcal{H} \vdash \mathrm{sp} \downarrow \mathrm{V}', \mathcal{H}' \quad \mathrm{X} \in \mathcal{D}(\mathrm{V}') \quad \mathrm{V}'(\mathrm{X}) = \mathrm{V}}{\mathcal{E}, \mathcal{H} \vdash \mathrm{sp.X} \downarrow \mathrm{V}, \mathcal{H}'} \ (4)$$

$$\frac{\mathcal{E}, \mathcal{H} \vdash \mathrm{sp} \downarrow \mathrm{ex}, \mathcal{H}'}{\mathcal{E}, \mathcal{H} \vdash \mathrm{sp.X} \downarrow \mathrm{ex}, \mathcal{H}'} \ (5)$$

$$\boxed{\mathcal{E}, \mathcal{H} \vdash \mathrm{s} \downarrow \mathrm{R}, \mathcal{H}'} \qquad \vdots$$

$$\frac{(\mathcal{E}, \mathrm{X} = l), (\mathcal{H}, l = \bot) \vdash \mathrm{s} \downarrow \mathrm{V}, \mathcal{H}' \quad l \notin \mathcal{D}(\mathcal{H})}{\mathcal{E}, \mathcal{H} \vdash \mathbf{rec}(\mathrm{X{:}S})\mathrm{s} \downarrow \mathrm{V}, (\mathcal{H}', l = \mathrm{V})} \ (6)$$

$$\frac{(\mathcal{E}, \mathrm{X} = l), (\mathcal{H}, l = \bot \vdash \mathrm{s} \downarrow \mathrm{ex}, \mathcal{H}') \quad l \notin \mathcal{D}(\mathcal{H})}{\mathcal{E}, \mathcal{H} \vdash \mathbf{rec}(\mathrm{X{:}S})\mathrm{s} \downarrow \mathrm{ex}, \mathcal{H}'} \ (7)$$

$$\frac{\begin{array}{ll} \mathrm{F} \in \mathcal{D}(\mathcal{E}) & \mathcal{E}(\mathrm{F}) = <\mathrm{X}, \mathcal{E}, \mathrm{s}'> \\ \mathcal{E}, \mathcal{H} \vdash \mathrm{s} \downarrow \mathrm{V}, \mathcal{H}' & (\mathcal{E}, \mathrm{X} = \mathrm{V}), \mathcal{H}' \vdash \mathrm{s}' \downarrow \mathrm{R}, \mathcal{H}'' \end{array}}{\mathcal{E}, \mathcal{H} \vdash \mathrm{F(s)} \downarrow \mathrm{R}, \mathcal{H}''} \ (8)$$

$$\frac{\mathrm{F} \in \mathcal{D}(\mathcal{E}) \quad \mathcal{E}(\mathrm{F}) = <\mathrm{X}, \mathcal{E}, \mathrm{s}'> \quad \mathcal{E}, \mathcal{H} \vdash \mathrm{s} \downarrow \mathrm{ex}, \mathcal{H}'}{\mathcal{E}, \mathcal{H} \vdash \mathrm{F(s)} \downarrow \mathrm{ex}, \mathcal{H}'} \ (9)$$

Figure 8: evaluation judgements

Although we do not attempt it here, we should be able to prove type soundness for (an instrumented version of) this semantics using the technique of Tofte [18] and its refinement by Elsman [6]. The main technical challenge in proving the type soundness result has less to do with the presence of the heap, which is similar to ML's store and should succumb to known proof techniques. The difficulty lies with ML's treatment of datatypes as *named* recursive types. Although type generativity ensures that each new datatype is assigned a unique type variable, a priori, there is nothing in the static semantics that prevents an existing datatype variable from being associated with more than one constructor environment in the context, making it impossible to deduce the structure of a constructed value from its type. Type soundness can still be proved modulo an interpretation of datatype variables as recursive types that is consistent with each type structure in the context (cf. [6]). The proof will appear in a future technical report.

6. SEPARATE COMPILATION

It is a simple exercise to show that the definition of the recursive signature and structure in Figures 3 and 4 typecheck. Although the definition of Eval contains two mutually recursive structures, it is not clear that we have gained very much. We have obtained the desired modular structure, but the substructures are still defined simultaneously, and, in their present form, cannot be typechecked or compiled in isolation. Support for separate compilation is one of the main motivations for introducing a module system. If the cost of programming with recursive modules is that their substructures must be defined in a single compilation unit, then, aside from better control of the namespace, we have not progressed much beyond the original restriction that confines recursive definitions to a single module.

```
(* unit EVAL.sig *)
signature EVAL =
  rec(X: sig structure Nat: sig type t end
             structure Bool: sig type t end
         end)
  sig structure Nat: sig
       datatype t = Zero | Succ of t
                  | If of X.Bool.t * t * t
       val eval: t -> t
      end
      structure Bool: sig
       datatype t = True | False | Null of X.Nat.t
       val eval: t -> t
      end
  end;
(* unit NatFun.sml *)
functor NatFun(X:EVAL) = struct
   datatype t = datatype X.Nat.t
   fun eval(n:t):t = case n : t of
      Zero => n | Succ m => Succ(eval m):t
    | If b t e => case X.Bool.eval b : X.Bool.t of
                   True => eval t
                 | False => eval e
end;
(* unit BoolFun.sml *)
functor BoolFun(X:EVAL) = struct
   datatype t = datatype X.Bool.t
   fun eval(b:t):t = case b : t of
         True => b | False => b
       | Null n => case X.Nat.eval n : X.Nat.t of
                    Zero => True:t
                  | Succ m => False:t
end
```

Figure 9: separately compiled functors

Ideally, we would like to separately compile the definitions of Bool and Nat as functors, each parameterised by the implementation of the other (Figure 9), and then take their fixed point in a separate compilation unit. The functors BoolFun and NatFun are completely independent and can be compiled separately (the common signature EVAL is a convenient abbreviation only, and could be in-lined, and indeed made smaller, at each occurrence).

The difficulty arises when we try to take the naive fixed point of the two functor applications:

```
structure Eval = rec(X:EVAL) struct
  structure Nat = NatFun(X)
  structure Bool = BoolFun(X)
end
```

This definition of Eval typechecks, but raises $\perp$, since the body of Eval attempts to evaluate its forward declaration before converging to a value (functor application is strict).

```
structure Eval = rec(X:EVAL) struct
 structure EtaX = struct (*an eta-expansion of X*)
     structure Nat = struct
        datatype t = datatype X.Nat.t
        val eval = λn:t. X.Nat.eval n
     end
     structure Bool = struct
        datatype t = datatype X.Bool.t
        val eval = λb:t. X.Bool.eval b
     end
 end
 structure Nat = NatFun(EtaX)
 structure Bool = BoolFun(EtaX)
end
```

Figure 10: using eta-expansion to reach a fixed-point

Fortunately, we can still obtain the correct fixed point if we apply the functors to an eta-expansion of the forward declaration (Figure 10). While awkward, and slightly less efficient than a direct definition (Figure 4), this solution applies whenever the forward declared values have function types. Note that our primary goal was to support cross-module recursive *functions*, which satisfy this criterion.

Another alternative, that lets us get away with the naive definition and avoid eta-expansion, would be to change our dynamic semantics and bind *all* structure identifiers to locations into the heap; this modification has the effect of penalising all structure projections, not just those from recursively bound structures, and reduces the performance of ordinary Standard ML programs.

7. ADVANCED EXAMPLE

Aside from modular programming, recursive modules also have applications in the implementation of advanced algorithms and data structures. Chris Okasaki's excellent book [13] gives a pseudo code implementation of "bootstrapped heaps" that requires recursive structures. Figure 11 contains a simplified version of his construction. The example compiles in Moscow ML but relies on some features not formalised in Mini-SML (pattern matching and higher-order, applicative functors [11, 14, 16]). The Bootstrap functor takes an arbitrary primitive heap functor, F, and an ordered

```
signature ORD = sig type t val leq:t*t->bool end
signature HEAP = sig
  structure Elem: ORD
  type heap
  val empty: heap
  val insert: Elem.t * heap ->heap
  val merge: heap * heap -> heap
  val findMin: heap -> Elem.t option
end

functor Bootstrap(F:functor O:ORD ->
                    HEAP where type Elem.t = O.t)
                 (O:ORD): HEAP = struct
 signature BOOT= rec(X:sig structure Elem:ORD end)
  sig structure Heap: sig
        type heap = App.heap where App = F(X.Elem)
      end
      structure Elem: sig
        datatype t = E | H of O.t * Heap.heap
        val leq : t * t -> bool
      end
  end
 structure Boot = rec(X:BOOT) struct
     structure Elem = struct
        datatype t = datatype X.Elem.t
        fun leq (H (x, _), H (y, _))= O.leq (x, y)
     end
     structure Heap = F(Elem)
 end
 structure Elem = O
 datatype heap = datatype Boot.Elem.t
 val empty = E
 fun merge(E, h) = h | merge(h, E) = h
   | merge(h1 as H(x,p1), h2 as H(y,p2)) =
   if O.leq(x,y) then H(x,Boot.Heap.insert(h2,p1))
   else H(y,Boot.Heap.insert(h1, p2))
 fun insert(x,h) = merge (H(x,Boot.Heap.empty),h)
 fun findMin E = NONE | findMin (H(x,_)) = SOME x
end
```

Figure 11: Okasaki's "bootstrapped" heaps

type, `O`, and returns an improved implementation of heaps over `O`. The datatype `Boot.Elem.t` is the type of bootstrapped heaps. A bootstrapped heap is either empty, `E`, or a node, `H`(x,p), consisting of a *root* element x of type `O.t` and an `F`-constructed primitive heap p of bootstrapped heaps. The root x caches the minimum element amongst all those bootstrapped heaps contained in the primitive heap p. Bootstrapped heaps are ordered by `Boot.Elem.leq` with respect to their root values. Note the use of signature recursion to construct the type `Heap.heap` from the recursive application `F(X.Elem)` (which relies on the assumption that `F` is *applicative*). With these data structures in place, the construction uses bootstrapped heaps to represent heaps of `O.t` elements as follows. To merge two heaps we insert the underlying bootstrapped heap with the larger root into the underlying bootstrapped heap with the smaller root, using the `insert` operation on bootstrapped heaps (not heaps). To insert an element into a heap, we create a singleton heap from the element and merge it. The minimum element of a (non-empty) heap is just the root at the node of the underlying bootstrapped heap. This construction improves the running time of both `findMin` and `merge` to $0(1)$ worst-case time, assuming that the original heap `F` supports `insert` in $0(1)$ and `merge` and `findMin` in $0(\log n)$ worst-case time [13].

8. RELATED WORK AND CONCLUSION

For presentation purposes, we restricted our attention to an explicitly typed, monomorphic Core language, but the extension scales to full Standard ML [15], whose Core language supports parameterised types and polymorphic values. Our extension is available in the current release of Moscow ML [17]. Adding recursive structures to Standard ML immediately extends the language with (explicit) polymorphic recursion, since a recursive function may call itself through a forward reference, that can be specified to have the required polymorphic type. In combination with Moscow ML's first-class modules [16], recursive structures may be used to define (syntactically heavy-weight) encodings of class-based objects with virtual methods, using functors to capture code inheritance (but not subtyping). Much more compelling is the useful combination of Moscow ML's higher-order functors with recursive structures, that we illustrated in Section 7.

Moscow ML's implementation of recursive structures is unsophisticated. We currently make no attempt to avoid dynamic checks, and the cost of accessing a forward declaration is proportional to its depth in the forward signature. We do, however, depart from the naive dynamic semantics by compiling enrichment as a coercion to a pruned structure value: in particular, the heap allocated value for a forward declaration is a pruned version of the structure body's value, reducing the potential for space leaks. Another obvious optimization would be to flatten the representation of this value to support depth-independent, constant-time access for each component in the forward declaration. Although it is difficult to construct realistic benchmarks, the current cost of calling a recursive function through a forward declaration at depth 0 appears to be approximately 60% more expensive than an ordinary recursive call. Removing the dynamic check for definedness (when safe) reduces the figure to 40%. Another possible optimisation is to represent the forward reference as a cell storing, not a tagged value, but a function that initially returns $\perp$ and is updated with a function that returns the value of the forward declaration, thus caching the result of the check. Replacing the dynamic checks by calls to these functions may be less expensive.

The important observation that recursive signatures can be used to specify mutually recursive types that span module boundaries, while recursive structures should be used to define mutually recursive values, appears in the paper by Crary, Harper and Puri [2] (recently revisited in [3]). That work is more theoretical than ours, and presents recursive signatures and modules as an extension of the phase distinction calculus of [9]. The authors use pseudo Standard ML syntax for their examples but leave the integration with Standard ML to future work. Our work may be seen as a concrete design based on this proposal, but it does differ in some aspects. In [2], a recursive signature does not contain a forward signature: all types in the signature body are considered to be mutually recursive and must be fully transparent. We allow finer control of the recursion in signatures and support opaque types in the signature body. More importantly, in [2] the forward declaration in a recursive module determines the type of the entire phrase, while in our proposal the body of a recursive structure must only enrich the forward declaration, but the full complement of its components will still be accessible from the expression as a whole. Our construct is more practical for the programmer, who must only make a forward declaration for those components that require forward references in the body; the remaining components in the body will not be hidden by the forward declaration, and need not be specified. Allowing for the addition of ordinary subtyping to [2] does not remove the distinction between our construct and theirs. Another difference is that the system in [2] statically rejects recursive modules that may be undefined, while we allow such modules at the cost of an inexpensive run-time check, that must be performed only when recursion crosses a module boundary. To allow fixed-points of module expressions other than values, but at the same time guarantee definedness, [2] uses a refined type system that draws a distinction between *valuable* and possibly undefined expressions. This is a cleaner solution, but we prefer to pay the cost of the run-time check rather than burden the Standard ML programmer with supplying more accurate type information in signatures. An optimizing compiler can still track valuability internally to remove unnecessary heap indirections and dynamic checks.

Duggan and Sourelis's [4] "mixin modules", although related to this work, solve a more general problem: to allow the definition of *individual* ML functions and datatypes to span module boundaries. Their system is an extension of SML-style Modules that allows datatypes, functions and initialisation code defined in one mixin module to be extended by constructors, match rules and further initialisation code defined in another, using a new operator called mixin composition. Composition merges two mixins to produce a new mixin; a separate construct closes a mixin module to take the fixed-point of its components. In [5], the authors propose an additional construct that links a simultaneous declaration of mixins containing unrelated, but mutually recursive, components, supporting cross-module recursion of the kind considered here. In combination, these extensions are more expressive than our own, but they also require more significant changes to Standard ML and its implementations. The semantics of the linking construct are not spelled out in detail, making a comparison with our approach difficult.

More distantly related to our work are Flatt and Felleisen's "units" [7], a module language for Scheme that caters for cyclicly dependent and possibly dynamically linked "units": although the authors present a typed version of the language, they point out that it does not give a satisfactory treatment of ML style type sharing, which we do support. Closely related to that proposal are Ancona and Zucca's [1] mixin modules, and Wells and Vestergaard's [19] similar module calculus. In those systems, a module may only contain terms, not types. The goals of these systems are different from ours, but this simplification avoids the main difficulty with type checking recursive modules, namely accommodating mutually recursive, cross-module type definitions, which is the focus of this paper.

9. REFERENCES

[1] *Ancona, D., and Zucca, E.* 1999. A primitive calculus of mixin modules. In *Proc. Principles and Practice of Declarative Programming*, LNCS vol. 1702 , pages 69-72, Berlin.

[2] *Crary, K., and Harper, R., and Puri, S.* 1999. What is a recursive module? In *Proc. ACM SIGPLAN'99 Conf. on Programming Language Design and Implementation*, pages 50-63.

[3] *Dreyer, D., and Harper, R., and Crary, K.* 2001. Toward a Practical Type Theory for Recursive Modules. TR CMU-CS-01-112, Computer Science, Carnegie-Mellon University, March..

[4] *Duggan, D., and Sourelis, C.* 1996. Mixin modules. In *Proc. ACM SIGPLAN International Conference on Functional Programming*, pages 262-273, Philadelphia.

[5] *Duggan, D., and Sourelis, C.* 1998. Parameterized modules, recursive modules, and mixin modules. In *Proc. ACM SIGPLAN Workshop on ML*, pages 87-96, Baltimore.

[6] *Elsman, M.* 1999. Program Modules, Separate Compilation, and Intermodule Optimisation. PhD thesis. University of Copenhagen.

[7] *Flatt, M., and Felleisen, M.* 1998. Units: cool modules for HOT languages. In *Proc. ACM SIGPLAN'98 Conf. on Programming Language Design and Implementation*, pages 236-248.

[8] *Harper, R., Lillibridge, M.* 1994. A type-theoretic approach to higher-order modules with sharing. In *21st ACM Symp. Principles of Programming Languages*.

[9] *Harper, R, and Mitchell, J. C., and Moggi, E.* 1990. Higher-order modules and the phase distinction. In *17th ACM Symp. Principles of Programming Languages*

[10] *Leroy, X.*, 1994. Manifest types, modules, and separate compilation. In *21st ACM Symp. Principles of Programming Languages*, pages 109–122. ACM Press.

[11] *Leroy, X.*, 1995. Applicative functors and fully transparent higher-order modules. In *22nd ACM Symp. Principles of Programming Languages*.ACM Press.

[12] *Milner, R., and Tofte, M., and Harper, R., and MacQueen, D.* 1997. The Definition of Standard ML (Revised). MIT Press.

[13] *Okasaki, C.* 1998. Purely Functional Data Structures. Cambridge University Press.

[14] *Russo, C. V.* 1998. Types For Modules. PhD Thesis, LFCS, University of Edinburgh.

[15] *Russo, C. V.* 1999. Non-Dependent Types For Standard ML Modules. In *1999 Int'l Conf. on Principles and Practice of Declarative Programming.*

[16] *Russo, C. V.* 2000. First-Class Structures for Standard ML. In *Nordic Journal of Computing*, 7(4):348, Winter 2000.

[17] *Sestoft, P., and Romanenko, S., and Russo, C. V.* 2000. Moscow ML V2.00.

[18] *Tofte, M.* 1988. Operational Semantics and Polymorphic Type Inference. PhD thesis, Computer Science, University of Edinburgh.

[19] *Wells, M., and Vestergaard, R.* 2000. Equational reasoning for linking with first-class primitive modules. In *Programming Languages & Systems, 9th European Symp. Programming*, LNCS vol. 1782 , pages 412-428, Berlin.