Abstract

Generalized algebraic datatypes (GADTs) are a hot topic in the functional programming community. Recently we showed that object-oriented languages such as C^♯ and Java can express GADT declarations using Generics, but only some GADT programs. The addition of equational constraints on type parameters recovers expressivity. We now study this expressivity gap in more depth by extending an earlier translation from System F to C^♯ to handle GADTs. Our efforts reveal some surprising limitations of Generics and provide further justification for equational constraints.

1. Introduction

Functional programming languages such as Haskell and ML have long supported user-defined datatypes. A datatype declaration simultaneously defines a named type, parameterized by other types, and the means of constructing values of that type. For example, here is a Haskell datatype of binary trees parameterized on the type d of data and type k of keys stored in the nodes:

```haskell
data Tree k d = Leaf | Node k d (Tree k d) (Tree k d)
```

This definition implicitly defines two value constructors Leaf and Node with polymorphic types:

- Leaf :: Tree k d
- Node :: k → d → Tree k d → Tree k d

Notice how both term constructors have the fully generic result type Tree k d; there is no specialization of the type parameters to Tree. Conversely, any value of type Tree τ σ, for some concrete τ and σ, can either be a leaf or a node — the static type does not reveal which. Observe that all recursive uses of the datatype within its definition are Tree k d; this makes Tree a regular datatype.

The restrictions on ordinary parameterized algebraic datatypes (PADTs) can be relaxed in the following three ways, yielding generalized algebraic datatypes (GADTs):

1. The restriction that constructors all return 'generic' instances of the datatype can be removed. This feature defines GADTs.
2. The regularity restriction can be removed, permitting datatypes to be used at different instantiations within their own definition. Writing useful functions over such types requires polymorphic recursion: the ability to use a polymorphic function at different types within its own definition. C^♯, Java and Haskell allow this, ML does not.
3. A constructor can be allowed to mention additional type variables that may appear in its argument types but do not appear in its result type. These type arguments are hidden by the type of the constructed term and thus existentially quantified.

Most useful examples of GADTs make use of all three abilities. Consider the following type Exp t representing abstract syntax for expressions of type t, written in a recent extension of Haskell with GADTs [2, 3]:

```haskell
data Exp t where
  Lit :: Int → Exp Int
  Plus :: Exp Int → Exp Int → Exp Int
  Equals :: Exp Int → Exp Int → Exp Bool
  Cond :: Exp Bool → Exp a → Exp a → Exp a
  Tuple :: Exp a → Exp b → Exp (a, b)
  Fst :: Exp (a, b) → Exp a
```

All constructors except for Cond make use of feature (1), as their result types refine the type arguments of Exp; for example, Lit has result type Exp Int. All constructors except for Lit make use of feature (2), using the datatype at different instantiations in arguments to the constructor. Finally, Fst uses a hidden type b, thus making use of feature (3).

Why is this interesting? Consider this evaluator for expressions, defined by case analysis on values of type Exp t:

```haskell
eval :: Exp t → t
eval e = case e of
  Lit i → i — t = Int
  Plus e1 e2 → eval e1 + eval e2 — t = Int
  Equals e1 e2 → eval e1 == eval e2 — t = Bool
  Cond e1 e2 e3 → — t = a
    if eval e1 then eval e2 else eval e3
  Tuple e1 e2 → (eval e1, eval e2) — t = (a, b)
  Fst e → fst (eval e) — t = a
```

The fascinating thing about eval is that the compiler doesn’t reject it. Observe closely: each branch of the case expression returns a computation of a different type. The Lit branch returns an integer, the Equals branch returns a boolean, the Tuple branch returns a pair. In the ML type system, all the continuations of a case expression are required to have the same type and one would expect eval to be rejected as type-incorrect. In GADT Haskell, this requirement is subtly relaxed: each branch must, instead, merely have an appropriate type, given the type of its pattern and the type of the scrutinee.

Although probably unintentional, both C^♯ and Java Generics already support GADTs. Consider the C^♯ code in Figure 1. This is a straightforward encoding of the GADT Haskell datatype Exp t. Abstract syntax trees are represented using an abstract class of expressions, with a concrete subclass for each node type. The interpreter is implemented by an abstract Eval1 method in the expression class, overridden for each node type. Indeed, this is a subtle variant of the Interpreter design pattern. Observe how the type parameter of Exp is refined in subclasses; moreover, this refinement...
is reflected in the signature and code of the overridden \texttt{Eval} methods. For example, \texttt{Plus} has result type \texttt{int} and requires no runtime casts in its calls to \texttt{Eval()} and \texttt{e2.Eval()}. Not only is this a clever use of static typing, it is also more efficient than a dynamically-typed version, particularly in an implementation that performs code specialization to avoid boxing [7].

Just like our Haskell datatype, these \texttt{C}\# classes make use of all three features that characterize \texttt{GADTs}. Feature (1) is expressed by defining a subclass of a generic type that does not just propagate its type parameters through to the superclass. (\texttt{Plus} is a non-generic class that extends the particular instantiation \texttt{Exp\texttt{int}}.) Feature (2) corresponds to the existence of fields in the subclass whose types are unrelated instantiations of the generic type of the superclass. (\texttt{Tuple\texttt{<A,B>}} has a field of type \texttt{Exp\texttt{<A>}} but superclass \texttt{Exp\texttt{<Pair\texttt{<A,B>}>}}.) Feature (3) corresponds to the declaration of type parameters on the subclass that are not referenced in the superclass. (\texttt{Fat\texttt{<A,B>}} has superclass \texttt{Exp\texttt{<A>}}, hiding \texttt{B}).

Where the Haskell \texttt{eval} function uses case analysis on expressions, the \texttt{C}\# code for \texttt{Eval} uses virtual dispatch to select the override of \texttt{Eval} appropriate to the expression node. The \texttt{C}\# signature of \texttt{Eval} specified in the \texttt{Exp\texttt{T}} class is a function of the type parameter \texttt{T}. Because this parameter is instantiated differently in each subclass, the overrides of \texttt{Eval} receive different signatures, obtained by substituting the actual type argument specified for the superclass in place of the formal type parameter \texttt{T}. For instance, the signature for the method \texttt{Lit} \texttt{Eval} is obtained by applying the substitution \texttt{T} \rightarrow \texttt{Int} to the signature specified in the superclass, so the override must return an \texttt{Int}, even though its declaration in the superclass just returns a \texttt{T}. Similarly, the signature for the method \texttt{Tuple\texttt{<A,B>}.Eval} is obtained by applying the substitution \texttt{T} \rightarrow \texttt{Pair\texttt{<A,B>}}, so the override must return a \texttt{Pair\texttt{<A,B>}}.

Haskell’s technique for typechecking the \texttt{eval} example is rather different. Haskell checks a case by checking each branch of the case under some equational assumptions, derived from equating the type (here \texttt{Exp t}) of the scrutinee (\texttt{e}) with the formal result type of the constructor guarding the branch (\texttt{Exp\texttt{Int}}, \texttt{Exp\texttt{Bool}}, \texttt{Exp\texttt{a}}, \texttt{Exp\texttt{b}}, etc). In \texttt{eval}, the assumptions are the equations on \texttt{t} shown in comments in each branch. Thus all branches do return a \texttt{t}, but each branch is allowed to make and exploit its own assumptions about what \texttt{t} is, given the type of the constructor guarding that branch. In general, typing a case expression exploits equational properties of types. In this code, each equation happens to correspond to a substitution for \texttt{t}, so it’s perhaps not surprising that the example translates to \texttt{C}\#, where we can specialize \texttt{T} in each superclass.

For a more involved example, consider the following annotated Haskell function, \texttt{eq}, that tests equality of expression values:

\begin{verbatim}
  eq :: (Exp t, Exp t) -> Bool
  eq (this, that) =
    case this of
      Lit i ->
        t = Int
        case that of
          Lit j -> i == j
          False
      Tuple e1 e2 ->
        t = (a, b)
        case that of
          Tuple f1 f2 ->
            t = (c, d)

  when Haskell typechecks the outer branch for \texttt{Tuple}, it assumes the type equation \texttt{t} \equiv \texttt{(a, b)} and type assignment \texttt{e1 :: Exp a}, \texttt{e2 :: Exp b}. In the inner branch it assumes \texttt{t} \equiv \texttt{(c, d)} and \texttt{f1 :: Exp c} and \texttt{f2 :: Exp d} (generating fresh names for the type parameters to the \texttt{Tuple} constructor). Using transitivity to combine the equations on \texttt{t} it obtains \texttt{(a, b)} \equiv \texttt{(c, d)}, and from this, derives \texttt{a} \equiv \texttt{c} and \texttt{b} \equiv \texttt{d}, using the fact that the product type constructor (\texttt{\_\_\_}) is injective. Hence \texttt{Exp a} \equiv \texttt{Exp c} and similarly \texttt{Exp b} \equiv \texttt{Exp d}, which lets Haskell type-check \texttt{eq\texttt{e1, f1}} and \texttt{eq\texttt{e2, f2}}. This use of equational decomposition, exploiting the injectivity of type constructors, is crucial to the type-checking of \texttt{eq}. Type checking \texttt{eval} was much easier: all equations were of the form \texttt{t} \equiv \texttt{r} and there was no need to decompose constructed types.

Not let us try to translate the \texttt{eq} example to \texttt{C}\#. We add a virtual method \texttt{Eq} to \texttt{ExpT}, taking a single argument that of type \texttt{ExpT} and by default returning \texttt{false}. Since \texttt{eq} is a function on pairs that performs nested case analysis, we implement it in \texttt{C}\# by dispatching twice, first on \texttt{this}, to the code that overrides \texttt{Eq} and then on that, to code specific to the types of both \texttt{this} and \texttt{that} (see Figure 2).

Unfortunately, this na"{i}ve translation does not typecheck. The problem is the override for \texttt{Tuple\texttt{<C,D>}} in the \texttt{Tuple\texttt{<A,B>}} class. The overridden method knows that \texttt{T=Pair\texttt{<A,B>}} by superclass specification, but it does not know that \texttt{T=Pair\texttt{<C,D>}}, which holds at its one and only call-site. Instead, consequences of this additional equation, for instance that \texttt{Exp\texttt{<A>=Exp\texttt{<C>}}} and \texttt{Exp\texttt{<B>=Exp\texttt{<D>}}}, can only be asserted with casts, leading to the code in Figure 3. The crux of the problem is this: although superclass instantiations are propagated to overrides through subclass refinement, there is no way to constrain the type instantiation of the receiver of a virtual method. Here, the only caller of virtual method \texttt{Tuple\texttt{<Eq\texttt{<C,D>}} happens to use the particular method instantiation \texttt{C=A,B=D} on a receiver of type \texttt{Exp\texttt{T}=Exp\texttt{<Pair\texttt{<A,B>}}}.

But the virtual method cannot specify the call-site invariant, so
Figure 2. Equality on values, type incorrect

```java
public abstract class Exp<T> { ...
  public virtual bool Eq(Exp<T> that)
  { return false; }
  public virtual bool TupleEq<T, D>(Tuple<T, D> e)
  { return false; }
  public virtual bool LitEq(Lit e)
  { return false; }
}
```

Figure 3. Equality on values, using casts

```java
public class Tuple<A, B> : Exp<Pair<A, B>> { ...
  public override bool TupleEq<T, D>(Tuple<T, D> e)
  { return e.e1.Eq((object) this.e1) && e.e2.Eq((object) this.e2); }
}
```

Figure 4. Equality on values, using constraints

```java
public abstract class Exp<T> { ...
  public virtual bool Eq(Exp<T> that)
  { return false; }
  public virtual bool TupleEq<T, D>(Tuple<T, D> e)
  { return false; }
  public virtual bool LitEq(Lit e)
  { return false; }
}
```

the particular override cannot assume it. Instead, it must assert the otherwise derivable type equalities using casts.

In [7], we propose extending C^5 to support *equality type constraints* on methods, as statically checked pre-conditions. Then adding the constraint where T=Pair<C, D> to the signature of TupleEq allows us to restrict its callers, and so avoid any casts (Figure 4). By instantiation of the superclass, the signature of the TupleEq override inherits the specialized constraint where Pair<A, B>=Pair<C, D>. From this, using a decomposition rule (this time, for C^5’s constructed types), one can derive A=C and B=D and finally Exp<A>=Exp<C> and Exp<B> = Exp<D>. It is these last two equations that justify the recursive calls to the Eq method on the fields of this and e. Here we rely on both method specialization in the subclass, which instantiates the override’s signature and its implicitly inherited equational constraint, and equational reasoning, to exploit equalities that flow from that specialised constraint.

A casual reader might object that equations are superfluous because one can instead directly define an equality method on the class Tuple<A, B> that lets one compare another Tuple<A, B> to this. But this would be missing the point: in order to call this method from the TupleEq<, > override in the Tuple<A, B> class, one would first have to have established that Tuple<A, B> = Tuple<C, D>, again requiring equational reasoning on types or an assertion using a cast. Note that we are trying to capture the rather general Haskell function that compares two expressions of the same, but otherwise unknown, expression type, not a family of functions that, for each particular form of expression, compares two instances of the same form and type.

Clearly, there is an expressivity gap between ordinary C^5 and GADT Haskell. In [7] we show that many interesting Haskell examples translate well, but some practically interesting ones, like Weirich’s type reconstruction algorithm taking untyped expressions to type expressions [3, 7], or the type safe LR parsers of Potter and Régis-Gianas [7] do not. These all fail for reasons exemplified by our contrived, but small, Eq operation. Roughly speaking, C^5 can express all of the datatypes of GADT Haskell, but only some of its programs — which ones? C^5 extended with equational constraints can express more GADT programs — but does it capture all of them? The aim of this paper to provide tentative answers to these questions, by studying translations from GADT variants of System F into C^5 and, separately, into C^5 with equational constraints.

The structure of this paper is as follows. We start by presenting our object of study, a featherweight version of C^5, called C^5 minor (Section 2). We then present our first variant of System F which we call G minor (Section 3). G minor employs a case construct that refines the types of branches using substitution only and is inspired by C^5’s typing of virtual methods and their overrides. We present a cast-free, type preserving translation from G major to C^5 minor, proving that C^5 minor is a least as expressive as G minor. Although quite natural, G minor has its own expressivity problems, requiring higher-order encodings to express some simple programs over PADTs (Section 3.2). This limitation of G minor manifests itself as a weakness in the design of both C^5 and Java Generics, that has, surprisingly, gone unnoticed in the literature (we believe that [2] is the first to make this observation). We then present G major, our second variant of System F (Section 4). G major is like G minor, but employs a more general typing rule for case that additionally derives equations particular to each case branch and adds an equational theory on types. G major incorporates the typing rule for case actually used in GADT Haskell and other studies of GADTs. Finally, we present C^5 major, an extension of C^5 minor with equational constraints on both methods and classes, and an equational theory on types (Section 5) (allowing constraints on classes is a slight improvement over [1]). Like the system in [7], C^5 major both fixes the observed defect in C^5 minor, and extends the range of expressible GADT programs. We present a cast-free, type preserving translation from G major to C^5 major demonstrating that this variant of C^5 major is at least as expressive as G major. Finally, in Section 6, we sketch a way of transforming programs of a certain kind in G major into equivalent ones in G minor.

The languages and translations described in the paper are summarised by the diagram:

```
G minor (§3) ⊂ ⊂ G major (§4)
G major (§5)
C^5 minor (§2) ⊂ ⊂ C^5 major (§5)
```

Figure 5. Expressivity hierarchy

2. C^5 minor

Our target language ‘C^5 minor’ [7] is a small, purely-functional subset of C^5 version 2.0 [7]. Its syntax, typing rules and big-step evaluation semantics are presented in Figures 5 and 6. To conserve space, the figures also present C^5 major with additions to C^5
Syntax:

\[
\begin{align*}
\text{(class def)} & \quad cd & ::= & \text{class } C < \Xi > : I \ \text{where } E \ \{ T \bar{f} ; kd. md \} \\
\text{(const def)} & \quad kd & ::= & \text{public } C(\bar{f} T) : \text{base}(\bar{f}) \ [\text{this.}\bar{f} = \bar{T}] \\
\text{(method qualifier)} & \quad Q & ::= & \text{public virtual} \ | \ \text{public override} \ \\
\text{(method def)} & \quad md & ::= & Q T m < X > T \ (T \bar{f}) \ \text{where } E \ [\text{return } e] \\
\text{(expression)} & \quad e & ::= & x \ | \ e.f \ | \ e.m < X > T | \ \text{new } I (T) \ | \ (T | T | T) \\
\text{(value)} & \quad v, w & ::= & \text{new } I (T) \ | \ (T | T | T) \\
\text{(type)} & \quad T, U, V & ::= & X | I \\
\text{(instantiated type)} & \quad I & ::= & C < T > \\
\text{(equational constraint)} & \quad E & ::= & I = U \\
\text{(typing environment)} & \quad \Gamma & ::= & X, \bar{f} : T, T \\
\text{(method signature)} & \quad \Gamma & ::= & X \bar{f} : T \rightarrow T \ (X \text{ is bound in } E, T, T) \\
\end{align*}
\]

Type Equivalence:

\[
\begin{align*}
\text{(ty-cast)} & \quad \Gamma \vdash T = U & \rightarrow & \Gamma \vdash T = U \\
\text{(ty-sub)} & \quad \Gamma \vdash T \rightarrow U & \rightarrow & \Gamma \vdash T \rightarrow U \\
\text{(ty-new)} & \quad \Gamma \vdash T = V & \rightarrow & \Gamma \vdash T = V \\
\text{(ty-meth)} & \quad \Gamma \vdash T = C < T > & \rightarrow & \Gamma \vdash C < T > = C < U > \\
\end{align*}
\]

Subtyping:

\[
\begin{align*}
\Gamma \vdash T = U & \rightarrow & \Gamma \vdash T < U \\
\Gamma \vdash T < U & \rightarrow & \Gamma \vdash T < V \\
\Gamma \vdash X < \text{object} & \rightarrow & \Gamma \vdash X < \text{object} \\
\end{align*}
\]

Well-formed contexts and types:

\[
\begin{align*}
\text{(ok-class)} & \quad \bar{T} = U \rightarrow \Gamma \vdash T \rightarrow U \\
\text{(ok-inst)} & \quad \bar{T} = U \rightarrow \Gamma \vdash T = U \\
\end{align*}
\]

Typing:

\[
\begin{align*}
\text{(ok-var)} & \quad \Gamma, x : T \vdash x : T \\
\text{(ok-fld)} & \quad \Gamma \vdash e : I \ \text{fields}(I) = T \bar{f} \\
\text{(ok-new)} & \quad \Gamma \vdash I \ \text{ok} \ \text{fields}(I) = T \bar{f} \\
\text{(ty-var)} & \quad \Gamma \vdash T = U \\
\text{(ty-fld)} & \quad \Gamma \vdash e.f_i : T_i \\
\text{(ty-new)} & \Gamma \vdash I \ \text{ok} \ \text{fields}(I) = T \bar{f} \\
\text{(ty-sub)} & \quad \Gamma \vdash T \rightarrow U \\
\text{(ty-meth)} & \quad \Gamma \vdash e.m < X > T \bar{f} : \{ T \bar{f} \} U \\
\end{align*}
\]

Method and Class Typing:

\[
\begin{align*}
\text{(ok-virtual)} & \quad \bar{T} = U \rightarrow \Gamma \vdash T \rightarrow U \rightarrow \text{ok} \\
\text{(ok-class)} & \quad \bar{T} = U \rightarrow \Gamma \vdash T \rightarrow U \rightarrow \text{ok} \\
\end{align*}
\]

**Figure 5.** Syntax and typing rules for C^5 minor (including highlighted changes for C^5 major)
minor highlighted. For C^5 minor, the additions should be treated as whitespace and ignored.

This formalisation is based on Featherweight GJ [?] and has similar aims: it is just enough for our purposes but does not “cheat” – valid programs in C^5 minor really are valid C^5 programs. The differences from Featherweight GJ are as follows:

- There are minor syntactic differences between Java and C^5; the use of ‘:’ in place of extends, and base in place of super. Methods must be declared virtual explicitly, and are overridden explicitly using the keyword override.
- For simplicity, we omit bounds on type parameters.
- We include a separate rule for subsumption instead of including subtyping judgments in multiple rules.
- We fix the evaluation order to be call-by-value.

Like Featherweight GJ, this language does not include object identity and encapsulated state, which arguably are defining features of the object-oriented programming paradigm. It does include dynamic dispatch, generic methods and classes, and runtime casts. For readers unfamiliar with the work on Featherweight GJ we summarise the language here; for more details see [2].

A **type** (ranged over by T, U and V) is either a formal type parameter (ranged over by X and Y) or the type instantiation of a class (ranged over by C, D) written C<T> and ranged over by I; object abbreviates object<>.

A **class definition** cd consists of a class name C with formal type parameters X, base class (superclass) I, constructor definition kd, typed instance fields T<T> and methods md. Method names in md must be distinct i.e. there is no support for overloading.

A **method qualifier** Q is either public virtual, denoting a publicly-accessible method that can be inherited or overridden in subclasses, or public override, denoting a method that overrides a method of the same name in some superclass.

A **method definition** md consists of a method qualifier Q, a return type T, name m, formal type parameters X, argument names T and types T, and a body consisting of a single statement return e;

A **constructor** kd simply initializes the fields declared by the class and its superclass.

An **expression** e can be a method parameter x, a field access e.f, the invocation of a virtual method at some type instantiation e.m<T>(T) or the creation of an object with initial field values new I(T). A value v is a fully-evaluated expression, and (always) has the form new I(T).

A **class table** D maps class names to class definitions. The distinguished class object is not in the table and treated specially.

A typing environment Γ has the form Γ = X, T where free type variables in T are drawn from X. We write ⊢ to denote the empty environment.

All of the judgment forms and helper definitions of Figures 5 and 6 assume a class table D. When we wish to be more explicit, we annotate judgments and helpers with D. We say that D is a valid class table if e."cd ok for each class definition cd in D and the class hierarchy is a tree rooted at object (not formalised here).

The operation mdty(T,m), given a statically known class T ≡ C<T> and method name m, looks up the generic signature of method m, by traversing the class hierarchy from C to find its virtual definition.

The operation mbody(T,m<T>,T), given a runtime class T ≡ C<T>, method name m and method instantiation T, walks the class hierarchy from C to find the most specific override of the virtual method, returning its body instantiated at types T.

**Theorem 1** (C^5 minor evaluation preserves typing). Suppose D is valid and ⊢ e : T. If e ⊢_{D} v then ⊢_{D} v : T.
3. System F with GADTs

In previous work \[?\] it was shown how a variant of System F, also known as the polymorphic lambda calculus, can be translated in a type-preserving way into \( C^9 \) minor. This demonstrated that generics in the style of \( C^6 \) and Java is as expressive as the impredicative ‘first-class’ polymorphism of System F.

In this section we extend System F with a (weak) form of GADTs, and exhibit an extended translation into \( C^9 \) minor. Following the musical theme, we call this language G minor. In addition to GADTs, it features polymorphic recursion for function values, typically required by GADT-manipulating programs. The syntax, typing rules and big-step evaluation semantics of G minor are presented in Figure 7. To conserve space, the figures also present G major with additions to G minor highlighted. For G minor, the additions should be treated as whitespace and ignored. Although somewhat artificial, we start with G minor because it is both natural and simpler than G major, yet has not been described in the literature on GADTs.

A typing environment \( \Gamma = \langle X, \pi, A \rangle \), consists of sequences of type variable declarations, and type assignments to term variables. An environment is \textit{valid} when \( X \) are distinct, \( \pi \) are distinct and all type variables free in \( \pi, A \) are drawn from \( X \). A typing judgment \( \Gamma \vdash M : \Delta \) should be read “in the context of a typing environment \( \Gamma \) the term \( M \) has type \( \Delta \)” with free type variables in \( M \) and \( \Delta \) drawn from \( \Gamma \). An evaluation judgment \( M \Downarrow^* V \) should be read “closed term \( M \) evaluates to produce a closed value \( V \)”.

We assume the presence of a global set of datatype declarations, defined by a finite set of type constructors \( T \), a function \( \Sigma \) mapping each type constructor \( D \in T \) to a finite set of term constructors \( K_D \), and term constructor \( k \in K_D \) to a type of the form \( \forall X \beta_A \). \( A \rightarrow DA \). We assume that each type constructor \( D \) takes a fixed number arity \( \beta(D) \) \( \geq 0 \) of type arguments, that all constructor types are \textit{closed}, and that the term constructors of distinct datatypes are disjoint, i.e. \( \forall X \beta_D \cap \forall X \beta_D' = \emptyset \), when \( D \neq D' \).

We identify types and terms up to renaming of bound variables, and assume that names of variables are chosen so as to be different from names already bound by \( \Gamma \). The notation \( [B/X]A \) denotes the capture-avoiding substitution of \( B \) for \( X \) in \( A \); likewise for \( [B/A]M \) and \( [V/x]M \).

Although there are no base types in G minor, booleans, natural numbers, and pairs can be encoded in the usual way; examples will be sugared using such types and operations. Observe that function values are polymorphic and recursive; moreover, functions can be used polymorphically within their own definition.

Consider the introduction and elimination rules for GADTs. The introduction rule (inj) is straightforward: it simply states that a term \( k X N \) is typed as if \( k \) were a polymorphic function (compare the rule for function application). The elimination rule (case) is more subtle. The case annotation \( (X : \gamma) \) for function application). The elimination rule (case) is more subtle. The

Choosing the case annotation \( (X : \gamma) \), we can implement eval using substitution based refinement of each branch’s type (just as in \( C^3 \) minor):

\[
\text{rec eval} = \Lambda X. \lambda x : \text{Exp} X : X.
\]

\[
\text{case } (X : \gamma) x \text{ of }
\]

\[
\begin{align*}
\text{Lit } y & \Rightarrow y \\
\text{Plus } y & \Rightarrow \text{eval int } (\pi_1 y) + \text{eval int } (\pi_2 y) \\
\text{Equals } y & \Rightarrow \text{eval int } (\pi_1 y) = \text{eval int } (\pi_2 y) \\
\text{Cond } Y y & \Rightarrow \\
\text{if eval bool } (\pi_1 y) \text{ then eval } Y (\pi_2 y) \text{ else eval } Y (\pi_3 y) \\
\text{Fst } Y Z y & \Rightarrow \pi_1 (\text{eval } Y \times Z y) \\
\text{Tuple } Y Z y & \Rightarrow \text{eval } Y (\pi_1 y), \text{eval } Z (\pi_2 y)
\end{align*}
\]

It straightforward to prove that evaluation preserves types.

**Theorem 2.** If \( \vdash M : A \) and \( M \Downarrow^* V \) then \( \vdash V : A \).

**Proof.** Induction on the evaluation derivation, using the usual Substitution and Weakening Lemmas.

3.1 Translation to \( C^9 \) minor

We now show how G minor programs can be translated to \( C^9 \) minor, thus demonstrating that \( C^6 \) can express at least the form of GADTs supported by G minor. The translation is based on an earlier translation from System F \[?\]; in particular, it uses a similar scheme for translating polymorphic functions.

Figure 8 presents the scheme for translating a G minor type \( A \) to a \( C^9 \) minor type \( A^* \), together with global class definitions \( G \) used in the translation.

A polymorphic function type \( \forall X. (A \rightarrow B) \) is translated into a type-instantiation of a named function class, whose single polymorphic method \( \text{app}<C> \) takes an argument of type corresponding to \( A \) and result type corresponding to \( B \). Function values are translated to instances of closure classes that extend the appropriate function class, in which the closure class is parameterized by the type parameters from the environment, the instance fields of the closure class store variables from the environment, and the body of the function is a method in the class that implements the app method. Recursion is translated into self-reference through \( \text{this} \). Function application is translated simply as invocation of the app method.

Parameterized datatypes are translated to parameterized classes, with one subclass for each constructor, as described informally in Section 1.

We require that the translation of types commute with substitute on type parameters. This forces the translation of an open type such as \( \forall X. (X \rightarrow Y) \) to be an instantiation of the same class as the translation of substitution instances such as \( \forall X. (X \rightarrow \text{int}) \).

In general, polymorphic types whose type variables appear at the same position in the types’ structure should translate to instantiations of the same named class. To achieve this we make use of an operation that Odersky and L" aufer call “lifting” \[?\].

**Definition 1** (Lifting). The \( \lambda \)-lifting of a \( C^6 \) minor type \( T \) is a pair \( (Y, T) \) in which \( Y \) is the abstracting out of maximal subterms \( T \) of the body that does not contain any \( X \), replacing the subterms by type variables \( Y \) such that \( T = \overline{T[Y/X]} \).

For example, the \( X \)-lifting of type \( \text{Fun}<\text{Fun}<X, Y>, \text{Fun}<Y, Y>> \) is the type abstraction \( \langle Z1, Z2\rangle \text{Fun}<\text{Fun}<X, Z1>, \text{Fun}<Y, Y>> \) together with the types \( Y \) and \( \text{Fun}<Y, Y> \) when substituted for variables \( Z1 \) and \( Z2 \) produce the original type.

The translation satisfies two important properties. First, it does not lose any type information, justifying the term “fully type-preserving”. Second, it commutes with substitution.

**Lemma 1.** \( A^* \equiv B^* \iff A = B \).
Proof. Easy induction on structure of types, using the identification of Fun types up to renaming of type variables.

\[ \text{Lemma 2. } ([B/X]A)^* = [B^*/X]A^* . \]

Proof. Similar to [?].

Figure 8 defines the translation of terms. The translation of a function \( \tilde{M} \) is given by a judgment

\[ \Gamma ; \psi \vdash^C M : A \rightsquigarrow e \in D \]

which should be read "In the context of typing environment \( \Gamma \) and argument environment \( \psi \) function \( M \) with type \( A \) translates to an expression \( e \) and additional class definitions \( D \) using fresh class names prefixed by \( C \)".

When translating the body \( M \) of a function value \( \text{rec } y = \Lambda X. \lambda (x:A):B.M \) it is necessary to distinguish three kinds of variable: the argument \( x \), the function \( y \) itself, or a free variable of the function. Likewise, when translating the branches of case constructs it is necessary to distinguish constructor arguments from free variables. To capture this in the translation the context contains both an ordinary typing environment \( \Gamma \) and argument environment \( \psi \) defined by the grammar

\[ \psi ::= y(X,x:A):B \mid k \bar{X} (x:A) \]

in which \( y \bar{X} (x:A):B \) denotes an environment used when translating functions in which \( x:A \) is the argument and \( y \bar{X} (A \to B) \) is the function, and \( k \bar{X} (x:A) \) denotes a constructor environment for constructor \( k \) in which \( \bar{X} \) are the type parameters to the constructor, and \( x:A \) is the constructor argument with its type. A similar split-context technique is used in treatments of typed closure conversion for functional languages [2].

The translation of function abstractions and case makes use of an operation \( \Gamma \uplus \psi \) that pushes the bindings from \( \psi \) into \( \Gamma \). It is
Global class definitions ($\mathcal{G}$):

For each $D \in \mathcal{T}$ define
\[
\text{class } D<\mathcal{T} : \text{object } \{
\}
\]
For each $k \in K_D$ with $\Sigma(D)(k) = \forall x A_k (A_k \rightarrow D_A)$ define
\[
\text{class } Dk<\mathcal{T}_k : D_Ak > \{ A_k; \text{ public } Dk(A_k, x) \{ \text{this}.x = x; \} \}
\]
For all $\mathcal{X}, T, U$ where freevars($T$) $\setminus \mathcal{X} = \mathcal{Y}$ and freevars($U$) $\setminus \mathcal{X} = \mathcal{Z}$ and identified up to renaming of $\mathcal{Y}\mathcal{Z}\mathcal{U}$ define
\[
\text{class } \text{Fun}_{\mathcal{X}(\mathcal{T})T\rightarrow U}<\mathcal{Y}\mathcal{Z}\mathcal{U} > \{ \text{public virtual } U \text{ app}<\mathcal{X}(T) > \{ \text{return } \text{this}.\text{app}<\mathcal{X}(x); \} \}
\]

Types:
\[
X^* = X \quad (D \mathcal{T})^* = D<\mathcal{T} >
\]
\[
(\forall \mathcal{X}. (A \rightarrow B))^* = \text{Fun}_{\mathcal{X}(\mathcal{T})T\rightarrow U}<\mathcal{Y}\mathcal{Z}\mathcal{U} >
\]
where $\mathcal{X}$-lifting of $A^*$ is $\langle (\mathcal{T})^*, \mathcal{T} \rangle$ and $\mathcal{X}$-lifting of $B^*$ is $\langle (\mathcal{Z})^*, \mathcal{U} \rangle$

Terms:

\[
\begin{align*}
\text{(tr-argvar)} \quad \Gamma; y(\mathcal{X}, x;A):B \vdash_C x : A \leadsto x & \quad \text{(tr-funvar)} \quad \Gamma; y(\mathcal{X}, x;A):B \vdash_C y : \forall \mathcal{X}. (A \rightarrow B) \leadsto \text{this} \leadsto x \\
\text{(tr-funfree)} \quad \mathcal{X}, x;A, y(\mathcal{X}, x;A):B \vdash_C x : A \leadsto \text{this}.x & \quad \text{(tr-casefree)} \quad \mathcal{X}, x;A, k(\mathcal{X}, x;A) \vdash_C x : A \leadsto \text{this}.x \\
\text{(tr-casearg)} \quad \Gamma; k(\mathcal{X}, x;A) \vdash_C x : A \leadsto \text{this}.x & \quad \text{(tr-inj)} \quad \Sigma(D)(k) = \forall \mathcal{X}. (A \rightarrow B) \quad \Gamma; \psi \vdash_C (A/\mathcal{X}) A \leadsto e \in \mathcal{D} \\
\text{(tr-app)} \quad \Gamma; \psi \vdash_C ^{\text{c}1} M \vdash_C \mathcal{X}. (A \rightarrow B) \leadsto e \in \mathcal{D}_1 & \quad \Gamma; \psi \vdash_C ^{\text{c}2} N : (A/\mathcal{X}) A \leadsto e' \in \mathcal{D}_2 \\
\text{(tr-abs)} \quad \Gamma; \psi ; y(\mathcal{Y}, x;A):B \vdash_C ^{\text{c}1} M : B \leadsto e \in \mathcal{D}_0 & \quad \Gamma; \psi \vdash_C ^{\text{c}2} \mathcal{Y} : \mathcal{A} \rightarrow \mathcal{A} \\
\Gamma; \psi \vdash \mathcal{X}, x;A & \quad \text{class } C<\mathcal{X} : (\forall \mathcal{Y}. (A \rightarrow B))^* \\
\text{D} = \{ C \rightarrow \{ \mathcal{A}; \text{ public } C(\mathcal{A}) \{ \text{this}.\mathcal{A} = \mathcal{A}; \} \text{ public override } B^* \text{ app}<\mathcal{Y}(A^*) > \{ \text{return } e; \} \} \}
\end{align*}
\]

\[
\begin{align*}
\text{(tr-case)} \quad \Gamma; \psi \vdash ^{\text{c}0} M : \forall x A_k \rightarrow D B_k & \in K & \iff \Gamma; \psi \vdash ^{\text{c}0} M : D_B \leadsto e \in \mathcal{D}_0 \\
\Gamma; \psi \vdash ^{\text{case}} \mathcal{X} ; \mathcal{A} \rightarrow \mathcal{A} & \quad \{ \Gamma; \psi ; k(\mathcal{X}, x;A_k) \vdash ^{\text{c}0} M_k : [B/\mathcal{T}] B \leadsto e_k \in \mathcal{D}_k \}_{k \in K_D} \\
\Gamma ; \psi \vdash ^{\text{case}} \mathcal{X} ; \mathcal{A} \rightarrow \mathcal{A} & \quad \{ \Gamma; \psi ; k(\mathcal{X}, x;A_k) \vdash ^{\text{c}0} M_k : [B/\mathcal{T}] B \leadsto e_k \in \mathcal{D}_k \}_{k \in K_D} \\
\end{align*}
\]

\[
\begin{align*}
\Gamma ; \psi \vdash = \mathcal{X}, \mathcal{T}, \mathcal{A} & \quad \text{M of } (k \mathcal{X}, x, A_k \rightarrow M_k) \in K & \iff \Gamma ; \psi \vdash ^{\text{case}} \mathcal{X} ; \mathcal{A} \rightarrow \mathcal{A} & \quad \{ \forall x A_k \rightarrow M_k \}_{k \in K} \rightarrow \mathcal{D}_0 & \in \mathcal{D}_0 \\
\end{align*}
\]

\[
\begin{align*}
\text{D} = \{ D \rightarrow \text{ class } D<\mathcal{T} : \text{object } \{ \text{public virtual } B^* \text{ case}<\mathcal{X}(\mathcal{T}) > \{ \text{return } \text{this}.\text{case}<\mathcal{X}(x); \} \} \} \\
\text{Dk} \rightarrow \text{ class } Dk<\mathcal{T}_k : (D_Ak) > \{ \text{public override } (D_Bk/T) B^* \text{ case}<\mathcal{X}(\mathcal{T}) > \{ \text{return } e_k; \} \} \}
\end{align*}
\]

Figure 8. Translation of types and terms

defined as follows:

\[
\begin{align*}
\Gamma ; \psi ; y(\mathcal{X}, x;A):B & \quad = \mathcal{X}, \Gamma; x; A, y : \forall \mathcal{X}. (A \rightarrow B) \\
\Gamma ; \psi ; k(\mathcal{X}, x;A) & \quad = \mathcal{X}, \Gamma; x; A
\end{align*}
\]

The translation is essentially defined by induction over the structure of the typing derivation of a term; $\Gamma; \psi \vdash ^{\text{c}1} M : A \leadsto e \in \mathcal{D}$ is defined when $\Gamma; \psi ; y(\mathcal{X}, x;A):B \vdash_C x : A \leadsto \text{this}.x = x$.

Consider rules (tr-inj) and (tr-case) for GADT introduction and elimination. Datatype constructors are translated to a simple use of $\text{new}$ on the appropriate constructor class. The case construct is translated to a $\text{case}$ method in the datatype class itself, together with overriding methods in each subclass. The context is abstracted as parameters to the case method; notice how the refinement of result type in the branches maps directly to refinement of the signature in the overridden methods.

We prove that the translation preserves types.

**Theorem 3** (Translation preserves types). If $\psi ; y(\mathcal{X}, x;A):B \vdash_C x : A \leadsto e \in \mathcal{D}$ then $\mathcal{D} \cup \mathcal{G}$ is a valid class table and $\psi ; y(\mathcal{X}, x;A):B \vdash_C x : A^*$.

**Proof.** Similar to analogous theorem in [?].

Future work is a theorem that the translation preserves evaluation behaviour, proved using the techniques of [?].
3.2 An Anomaly of G minor and Object-Oriented Generics

It is both simple to state and prove sound, but there is something quite odd about G minor’s (case) rule. Although the type of the case scrutinee may be any instantiation \( \Sigma \) of the datatype, each branch must be completely parametric in its constructor’s formal type parameters (whether or not they have an existential flavour). As a result, writing instantiation-specific case expressions is extremely awkward. For example, a (non GADT) Haskell programmer would expect to be able to translate the simple (first-order) `sum` function, that adds the integers in a list, as follows:

\[
\text{Summing list of integers}
\]

As we saw in Section 1, there are functions over GADTs that cannot be expressed using a type function to refine the types of case branches. Instead, we can express the relationship between the scrutinee’s type and the range of the constructors through type equations, and then use these equations in the branch body. To this end, we extend G minor with equations, calling the extended language G major. The extensions are highlighted in Figure 7.

- The equational rules state that (a) type equality is a congruence (a reflexive, symmetric and transitive relation compatible with type formation), and (b) type constructors are injective. Injectivity is crucial to typing examples like `eq` from Section 1.

4. System F with GADTs and equations

As we saw in Section 1, there are functions over GADTs that cannot be expressed using a type function to refine the types of case branches. Instead, we can express the relationship between the scrutinee’s type and the range of the constructors through type equations, and then use these equations in the branch body. To this end, we extend G minor with equations, calling the extended language G major. The extensions are highlighted in Figure 7.

- The equational rules state that (a) type equality is a congruence (a reflexive, symmetric and transitive relation compatible with type formation), and (b) type constructors are injective. Injectivity is crucial to typing examples like `eq` from Section 1.

- We introduce a new term former \( M @ A \) and subsumption-like typing rule (eqn). This allows us to retyping \( M \) at a derivatively equivalent type. We could have used implicit subsumption, but terms would not then uniquely determine derivations, a property that makes the translations slightly easier to formalize. Pottier and Régis-Gianas [7] introduce a similar device.

- The (eqn-case) typing rule extends (case) from Figure 7 simply by introducing equations into the context for each branch. Each equation equates the formal instantiation \( \Gamma \) of the pattern with the actual instantiation \( M \) of the type of the scrutinee, potentially inducing more equations on both the type variables bound by the pattern \( X_i \) and the ambient type variables in \( \Gamma \).

When we wish to be explicit, we write \( \Gamma \vdash_{\text{min}} M : A \) for typing judgments in G minor, and \( \Gamma \vdash_{\text{max}} M : A \) for G major.
\[
\text{rec sum} = \lambda (x: \text{List} \text{Int}) : \text{Int. case } (x)\text{ of} \\
\text{Nil } y \Rightarrow 0 \\
\text{Cons } y \Rightarrow ((\pi_1 y) \text{@Int}) + \text{sum } ((\pi_2 y) \text{@List} \text{Int}) \quad \text{where } Y \equiv \text{Int}
\]

\[
\text{rec eq} = \lambda X. (\lambda (x: \text{Exp} \times \text{Exp} \times \text{Exp} \times \text{Exp}) : \text{bool. case } (X)\text{ of} \\
\text{Lit } y \Rightarrow \text{case } \pi_2 x\text{ of} \\
\text{Lit } z \Rightarrow z = z \\
\ldots \Rightarrow \text{false} \\
\text{Tuple } \text{YY} y \Rightarrow // X \equiv Y \times Y' \\
\text{case } \pi_2 x\text{ of} \\
\text{Tuple } \text{ZZ} z \Rightarrow // X \equiv Z \times Z' \\
\text{eq } Y (\pi_1 y, \pi_1 z) \text{@Exp} Y) \land \text{eq } Y' (\pi_2 y, \pi_2 z) \text{@Exp} Y') \\
\ldots \Rightarrow \text{false}
\]

**Figure 10.** Summing lists and equality values, in G major

In rule (eqn-case), if \(D\) is actually a PADT (\(\overline{D_k} \equiv \overline{X_k}\), for each \(k\)), then the rule degenerates to ordinary case over PADTs as found in vanilla Haskell and ML: the equations just instantiate \(X_k\).

Figure 10 presents the problematic sum and Eq functions from Sections 3.2 and 1, both written in G major. Notice the \(\pi\) terms, making use of the equations shown in comments, together with rules (tran) and (c1) and, for function eq only, rule (d1).

Inspecting the rules shows G major is a conservative extension of G minor.

**Lemma 3.** If \(\Gamma \vdash^\text{ minor} M : A\) then \(\Gamma \vdash^\text{ major} M : A\).

The use of type equations in the (eqn-case) rule recalls other presentations of GADTs \([?, ?, ?]\). However, G major retains the ability to refine the types of branches through a type function \((X)C\). When this type function is constant (i.e. \(X\) are not free in \(C\)) then refinement only occurs through equations. The following lemma shows that equations alone suffice.

**Lemma 4.** If \(\Gamma \vdash^\text{ major} M : A\) then there is a term \(N\) such that \(\Gamma \vdash^\text{ major} N : A\), whose type-erasure is identical to that of \(M\) and whose type function annotations are constant.

**Proof.** By induction on the typing derivation. (For eqn-case, suppose that we have a G minor term)

\[
\text{case } (X)C M \text{ of } \{ k: X_k \Rightarrow M_k \}_{k \in K}
\]

type \(C' = [B/\overline{X}]C\). Assume (by induction) that \(M\) transforms to \(M'\) and \(M_k\) transforms to \(M'_k\) for each \(k \in K\). Then we can construct a G major term

\[
\text{case } (X)C M' \text{ of } \{ k: X_k \Rightarrow M'\}_k \text{ at } C' \text{ using rule (eqn) and the equation } T \equiv T'\text{ from the context.}
\]

**Lemma 5** (Equation Elimination). Let \(J\) range over type formation, type equivalence and typing judgment forms \((A, A \equiv B \text{ and } M : A)\). If \(\Gamma, \overline{E} \vdash J\) and \(\Gamma \vdash \overline{E}\) then \(\Gamma \vdash J\).

**Proof.** Induction on the derivation of \(J\).

**Theorem 4** (Evaluation preserves typing). If \(\vdash M : A\) and \(M \downarrow^\text{ E} V\) then \(\vdash V : A\).

**Proof.** Induction on the evaluation derivation.

---

### 5. Adding equations to C^5

In Section 1 we observed that the Eq method on expressions cannot be typed without resorting to casts. We sketched how the addition of equational constraints on type parameters, together with some equational reasoning on types, allows us to avoid these casts. Here, we present a formalization of these ideas as an extension to C^5 minor, called C^5 major. For a more gentle exposition, see \([?]\). The syntax, typing and helper definitions of C^5 major are shown in Figures 5 and 6, but this time including the highlighted bits. In brief, C^5 major extends C^5 minor as follows:

- Class and virtual method declarations can specify sets of equations between types (typically involving class and method type parameters) as additional preconditions.
- Class constraints restrict the formation of constructed types to those whose type arguments satisfy the constraints.
- Contexts now contain sets of equations as well as type parameters and type assignments.
- A new equational judgement on types states that (a) type equality is a congruence, and (b) type constructors are injective.
- Internal method signatures, returned by the helper relation \(ntype\), may mention equations, inherited from the virtual declaration and possibly specialised through inheritance.
- The reflexivity rule for subtyping is extended to include derivably equal, not just identical types. The usual subsumption rule can now be used to re-type a term at a different, but equivalent, type (as well as catering for subtyping as usual).
- The typing rules for methods extend the ones from Figure 5 simply by introducing well-formed class and (possibly inherited) method constraints into the context of the method body.
- In turn, method constraints restrict legal method invocations to those that satisfy the constraints of both the enclosing instantiated class and the instantiated method itself (Rule (ty-meth)). The former condition is implicit in the premise \(\Gamma \vdash e : I\), since this implies \(\Gamma \vdash I\ ok\).

C^5 major’s support for equational constraints on classes extends \([?]\), which only allows for constraints on methods. We include this feature to enable the translation from G major. Our translation closure-converts a G major function into a C^5 major method, whose enclosing class must capture the translated context of the function. To preserve types, the class of this method must now record any equations in the G major context. This requires equationally constrained classes.

The key to proving type preservation for C^5 major is the following lemma, that allows one to discharge established equational hypotheses from typing judgments:

**Lemma 6** (Equation Elimination). Let \(J\) range over type equivalence, type formation, subtyping and typing judgment forms \((e : T, T = U \text{ and } T < : U)\). If \(\Gamma, \overline{E} \vdash J\) and \(\Gamma \vdash \overline{E}\) then \(\Gamma \vdash J\).

**Proof.** Induction on the derivation of \(J\).

In \([?]\), we prove a full Type Soundness theorem, combining Preservation and Progress, but here we content ourselves with:

**Theorem 5** (C^5 major evaluation preserves typing). Suppose that \(D\) is a valid class table and \(\vdash^D e : T\). If \(e \downarrow^D v\) then \(\vdash^D v : T\).
5.1 Translation from G major to C^s major

G major programs can be translated into C^s major programs, extending the translation of Figure 8. The new translation is shown in Figure 11, with the additions highlighted.

The argument environment \( \psi \) is extended with equations guarding a constructor; these equations are propagated into the context \( \Gamma \) through the \( \psi \) operation. The (tr-abs) rules closes over equations by declaring them in the closure class; analogously, the (tr-eq-case) rule closes over equations by declaring them on the case method. In addition, (tr-eq-case) declares an equation \( \uparrow \psi \) which gets refined in constructor subclasses to \( \uparrow \psi \ast \uparrow \psi \) as we require.

**Lemma 7.** Suppose that \( X, \Gamma \vdash T \uparrow U \) with \( X \) not free in \( \Gamma \), and that the \( \uparrow X \)-lifting of \( T \) is \( \langle (Y) V, T \rangle \). Then the \( \uparrow X \)-lifting of \( U \) is \( \langle (Y) V, U \rangle \) for some \( U \) such that \( X, \Gamma \vdash \uparrow T \uparrow U \).

**Proof.** By induction on the equality derivation.

**Lemma 8.** If \( \Gamma \vdash A \equiv B \) then \( \Gamma^* \vdash A^* \equiv B^* \).

**Proof.** By induction on the derivation. Most cases are straightforward, with case (c2) relying on Lemma 7.

**Theorem 6** (Translation preserves types). \( \downarrow \psi \vdash M : A \rightsquigarrow e \) in \( \mathcal{D} \) then \( \mathcal{D} \uparrow \Gamma \) is a valid class table and \( \downarrow \psi \vdash M \ast B : A^* \rightsquigarrow e^* \) in \( \mathcal{D} \).

**Proof.** As Theorem 3, with Lemma 8 used for rule (eqn).

5.2 Translation from G major to C^s minor

C^s minor does not support the equational constraints necessary to express G major terms using static typing. However, there is a translation that makes use of checked downcasts: for every use of (eqn), the translation inserts a cast. We simply change the (tr-eq-case) rule to be:

**Figure 3** illustrated this use of casts that is necessary in the absence of equational constraints.

6. From G major to G minor

We have observed how rule (case) in G minor, and the use of polymorphic inheritance in C^s minor, force case analysis over GADTs to be completely parametric in the type parameters of the datatype.
Equations, as featured in G major and C♯ major, provide a way out, expressing type specialization of the datatype. But we have also seen in Section 3.2 how in the case of ordinary datatypes, equations can be avoided, at the cost of introducing higher-order functions.

In general, when are equations required? It seems that the use of the decomposition rules, expressing the injectivity of type constructors, is crucial. Consider the following simple example:

$$\lambda x : D \text{ int. } \begin{cases} \text{case } (y : \text{bool}) \text{ of } \\ \quad k_1 \ x \ y \Rightarrow y@\text{int} = 5 \quad // X \equiv \text{int} \\ \quad k_2 \ z \Rightarrow z \end{cases}$$

At first glance, this function over a GADT appears to make essential use of the equation $X \equiv \text{int}$. However, it turns out that the term can be massaged a little to eliminate the use of $\emptyset$, by abstracting it out as a coercion whose type-erasure is the identity function:

$$\lambda x : D \text{ int. } \begin{cases} \text{case } (y : \text{int}) \rightarrow \text{bool} \text{ of } \\ \quad k_1 \ x \ y \Rightarrow \lambda f : (X \rightarrow \text{int}) . f \ y = 5 \\ \quad k_2 \ z \Rightarrow \lambda f : (\text{int} \rightarrow \text{int}) . z \end{cases}$$

In the general case, we believe that all uses of decomposition-free (eqn) can be abstracted out as functions passed through case.

**Conjecture 1.** Any decomposition-free use of (eqn) can be hoisted outside its nearest enclosing case term by a meaning-preserving transformation. If there is no further enclosing case then it can be eliminated completely.

Iterating this construction leads to the following corollary: for any G major term that has a decomposition-free derivation, there is a semantically-equivalent term in G minor.

### 7. Conclusion

We have characterized the ‘expressivity gap’ between GADT programs in C and GADT programs in functional languages by studying two extensions of System F, one using type functions to refine the typings of case branches, and the other using more powerful type equations. We proved that C minor (hence C) is at least as expressive as the weaker, first variant, and, once extended with equational constraints, at least as expressive as the second variant too.

We have not shown that our translations preserve evaluation (termination behaviour), but that should be possible using the techniques in [2]. In future, it would be nice to prove that there are G major (cast-free C) major) programs, such as $eq$ (Eq), that cannot be expressed in G minor (cast-free C minor), pinning down the expressivity of the weaker systems, but that would require more sophisticated methods.

We originally believed that C was as expressive as GADT Haskell. We only stumbled upon G minor by attempting to formulate a simple variant of System F with GADTs, suitable as a source language for our translation, but without the complications of nested pattern matching, constrained polymorphism, higher-kinded type variables etc. found in other presentations. We were surprised to find that G minor was too simple, and could not express some GADT programs expressible in Haskell. This is how we identified the problematic Eq example in C. It took a second look at the presentation of GADTs in the literature for us to fully appreciate the advantage of building in an equational theory on types, including the crucial decomposition rules. One conclusion that could be drawn from this work is that the anomaly of C minor (Section 3.2), which only became apparent to us when we formulated the case rule of G minor, reveals a flaw in the design of C and Java Generics. It seems odd for Generics to provide elegant support for, but not all, GADT programs, but rather poor support for programming with ordinary PADTs such as List<\text{T}>. Of course, one could also argue that instantiation specific operations, such as $\emptyset$, really have no place as virtual methods on their generic class, because the applicability of such operations is restricted. That would be fine, provided the language provided some alternative mechanism for instantiation specific case analysis. However, the absence of any other instantiation specific construct for (safely) dispatching on runtime types, and the fact that the workaround of resorting to the (non-extensible) visitor pattern conflicts with that other object-oriented goal of preserving subtype extensibility, leads us to conclude that Generics is deficient in this regard, and could be improved by the addition of equational constraints along the lines of C major. The observation that, without constraints, GADT’s do not admit Visitor patterns [2], also hints at an incompleteness in Generics that is remedied by our extension.

### References


