Transposing G to C$^\sharp$: Expressivity of Generalized Algebraic Data Types in an Object-Oriented Language

Andrew J. Kennedy
Microsoft Research Ltd, Cambridge, U.K.
akenn@microsoft.com

Claudio V. Russo
Microsoft Research Ltd, Cambridge, U.K.
crusso@microsoft.com

Abstract

Generalized algebraic datatypes (GADTs) are a hot topic in the functional programming community. Recently we showed that object-oriented languages such as C$^\sharp$ 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$^\sharp$ 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:

$$\mathbf{data}\ \mathit{Tree}\ k\ d\ =\ \mathit{Leaf}\ \mid\ \mathit{Node}\ k\ d\ (\mathit{Tree}\ k\ d)\ (\mathit{Tree}\ k\ d)$$

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

$$\begin{array}{l} \mathit{Leaf}\ ::\ \mathit{Tree}\ k\ d \\ \mathit{Node}\ ::\ k\ \to\ d\ \to\ \mathit{Tree}\ k\ d\ \to\ \mathit{Tree}\ k\ d\ \to\ \mathit{Tree}\ k\ d \end{array}$$

Notice how both term constructors have the fully generic result type $\mathit{Tree}\ k\ d$; there is no specialization of the type parameters to Tree. Conversely, any value of type $\mathit{Tree}\ \tau\ \sigma$, 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 $\mathit{Tree}\ k\ d$: 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$^\sharp$, 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 $\mathit{Exp}\ t$ representing abstract syntax for expressions of type t, written in a recent extension of Haskell with GADTs [**?**, **?**]:

$$\begin{array}{l} \mathbf{data}\ \mathit{Exp}\ t\ \mathbf{where} \\ \quad \mathit{Lit}\ ::\ \mathit{Int}\ \to\ \mathit{Exp}\ \mathit{Int} \\ \quad \mathit{Plus}\ ::\ \mathit{Exp}\ \mathit{Int}\ \to\ \mathit{Exp}\ \mathit{Int}\ \to\ \mathit{Exp}\ \mathit{Int} \\ \quad \mathit{Equals}\ ::\ \mathit{Exp}\ \mathit{Int}\ \to\ \mathit{Exp}\ \mathit{Int}\ \to\ \mathit{Exp}\ \mathit{Bool} \\ \quad \mathit{Cond}\ ::\ \mathit{Exp}\ \mathit{Bool}\ \to\ \mathit{Exp}\ a\ \to\ \mathit{Exp}\ a\ \to\ \mathit{Exp}\ a \\ \quad \mathit{Tuple}\ ::\ \mathit{Exp}\ a\ \to\ \mathit{Exp}\ b\ \to\ \mathit{Exp}\ (a,b) \\ \quad \mathit{Fst}\ ::\ \mathit{Exp}\ (a,b)\ \to\ \mathit{Exp}\ a \\ \quad \ldots \end{array}$$

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 $\mathit{Exp}\ \mathit{Int}$. 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 $\mathit{Exp}\ t$:

$$\begin{array}{l} \mathit{eval}\ ::\ \mathit{Exp}\ t\ \to\ t \\ \mathit{eval}\ e\ =\ \mathbf{case}\ e\ \mathbf{of} \\ \quad \mathit{Lit}\ i\ \to\ i\quad \text{— } t = \mathit{Int} \\ \quad \mathit{Plus}\ e1\ e2\ \to\ \mathit{eval}\ e1\ +\ \mathit{eval}\ e2\quad \text{— } t = \mathit{Int} \\ \quad \mathit{Equals}\ e1\ e2\ \to\ \mathit{eval}\ e1\ ==\ \mathit{eval}\ e2\quad \text{— } t = \mathit{Bool} \\ \quad \mathit{Cond}\ e1\ e2\ e3\ \to\quad \text{— } t = a \\ \qquad \mathbf{if}\ \mathit{eval}\ e1\ \mathbf{then}\ \mathit{eval}\ e2\ \mathbf{else}\ \mathit{eval}\ e3 \\ \quad \mathit{Tuple}\ e1\ e2\ \to\ (\mathit{eval}\ e1,\ \mathit{eval}\ e2)\quad \text{— } t = (a,b) \\ \quad \mathit{Fst}\ e\ \to\ \mathit{fst}\ (\mathit{eval}\ e)\quad \text{— } t = a \\ \quad \ldots \end{array}$$

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$^\sharp$ and Java Generics already support GADTs. Consider the C$^\sharp$ code in Figure 1. This is a straightforward encoding of the GADT Haskell datatype $\mathit{Exp}\ t$. 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 `Eval` 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

```
public abstract class Exp<T>
  { public abstract T Eval(); }
public class Lit : Exp<int> { int value;
  public Lit(int value) { this.value=value; }
  public override int Eval() { return value; }
}
public class Plus : Exp<int> { Exp<int> e1, e2;
  public Plus(Exp<int> e1, Exp<int> e2)
    { this.e1=e1; this.e2=e2; }
  public override int Eval()
    { return e1.Eval() + e2.Eval(); }
}
public class Equals : Exp<bool> { Exp<int> e1, e2;
  public Equals(Exp<int> e1, Exp<int> e2)
    { this.e1=e1; this.e2=e2; }
  public override bool Eval()
    { return e1.Eval() == e2.Eval(); }
}
public class Cond<T> : Exp<T> {
  Exp<bool> e1; Exp<T> e2, e3;
  public Cond(Exp<bool> e1, Exp<T> e2, Exp<T> e3)
    { this.e1=e1; this.e2=e2; this.e3=e3; }
  public override T Eval()
    { return e1.Eval() ? e2.Eval() : e3.Eval(); }
}
public class Tuple<A,B> : Exp<Pair<A,B>> {
  Exp<A> e1; Exp<B> e2;
  public Tuple(Exp<A> e1, Exp<B> e2)
    { this.e1=e1; this.e2=e2; }
  public override Pair<A,B> Eval()
    { return new Pair<A,B>(e1.Eval(), e2.Eval()); }
}
public class Fst<A,B> : Exp<A> { Exp<Pair<A,B>> e;
  public Fst(Exp<Pair<A,B>> e){ this.e=e; }
  public override A Eval(){ return e.Eval().fst; }
}
```

Figure 1. Typed expressions with evaluator

is reflected in the signature and code of the overridden `Eval` methods. For example, `Plus.Eval` has result type `int` and requires no runtime casts in its calls to `e1.Eval()` and `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 [**?**].

Just like our Haskell datatype, these C$^\sharp$ classes make use of all three features that characterize 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. (`Plus` is a non-generic class that extends the particular instantiation `Exp<int>`.) Feature (2) corresponds to the existence of fields in the subclass whose types are unrelated instantiations of the generic type of the superclass. (`Tuple<A,B>` has a field of type `Exp<A>` but superclass `Exp<Pair<A,B>>`.) Feature (3) corresponds to the declaration of type parameters on the subclass that are not referenced in the superclass. (`Fst<A,B>` has superclass `Exp<A>`, hiding `B`.)

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

Haskell's technique for typechecking the $eval$ 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 $Exp\ t$) of the scrutinee (e) with the formal result type of the constructor guarding the branch ($Exp\ Int$, $Exp\ Bool$, $Exp\ a$, $Exp(a, b)$ etc). In $eval$, the assumptions are the equations on t shown in comments in each branch. Thus all branches do return a t, but each branch is allowed to make and exploit its own assumptions about what 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 t, so it's perhaps not surprising that the example translates to C$^\sharp$, where we can specialize `T` in each superclass.

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

$$
\begin{array}{l}
eq\ ::\ (Exp\ t,\ Exp\ t)\ \rightarrow\ Bool \\
eq\ (this, that)\ = \\
\quad \textbf{case}\ this\ \textbf{of} \\
\qquad Lit\ i\ \rightarrow \qquad \text{---}\ t = Int \\
\qquad\quad \textbf{case}\ that\ \textbf{of} \\
\qquad\qquad Lit\ j\ \rightarrow\ i == j \quad \text{---}\ t = Int \\
\qquad\qquad \_\ \rightarrow\ False \\
\qquad Tuple\ e1\ e2\ \rightarrow \qquad \text{---}\ t = (a, b) \\
\qquad\quad \textbf{case}\ that\ \textbf{of} \\
\qquad\qquad Tuple\ f1\ f2\ \rightarrow \qquad \text{---}\ t = (c, d) \\
\qquad\qquad\quad eq\ (e1, f1)\ \&\&\ eq\ (e2, f2) \\
\qquad\quad \_\ \rightarrow\ False
\end{array}
$$

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

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

Unfortunately, this naive translation does not typecheck. The problem is the override for `TupleEq<C,D>` in the `Tuple<A,B>` class. The overriden method knows that `T=Pair<A,B>` by superclass specialisation, but it does not know that `T=Pair<C,D>`, which holds at its one and only call-site. Instead, consequences of this additional equation, for instance that `Exp<A>=Exp<C>` and `Exp<B>=Exp<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 `TupleEq<C,D>` happens to use the particular method instantiation `C=A,D=B` on a receiver of type `Exp<T>=Exp<Pair<A,B>>`. But the virtual method cannot specify the call-site invariant, so

```
public abstract class Exp<T> { ...
  public virtual bool Eq(Exp<T> that)
    { return false; }
  public virtual bool TupleEq<C,D>(Tuple<C,D> e)
    { return false; }
  public virtual bool LitEq(Lit e)
    { return false; }
}
public class Lit : Exp<int> { ...
  public override bool Eq(Exp<int> that)
    { return that.LitEq(this); }
  public override bool LitEq(Lit e)
    { return value == e.value; }
}
public class Tuple<A,B> : Exp<Pair<A,B>> { ...
  public override bool Eq(Exp<Pair<A,B>> that)
    { return that.TupleEq<A,B>(this); }
  public override bool TupleEq<C,D>(Tuple<C,D> e)
    { return e.e1.Eq(this.e1) && e.e2.Eq(this.e2); }
}
```

Figure 2. Equality on values, type incorrect

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

Figure 3. Equality on values, using casts

```
public abstract class Exp<T> { ...
  public virtual bool TupleEq<C,D>(Tuple<C,D> e)
    where T=Pair<C,D> { return false; }
}
public class Tuple<A,B> : Exp<Pair<A,B>> { ...
  public override bool Eq(Exp<Pair<A,B>> that)
    { return that.TupleEq<A,B>(this); }
  public override bool TupleEq<C,D>(Tuple<C,D> e)
    { return e.e1.Eq(this.e1) && e.e2.Eq(this.e2); }
}
```

Figure 4. Equality on values, using constraints

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

In [?], we propose extending C♯ to support *equational 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♯'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> that` to `this`. But this would be missing the point: in order to call this method from the `TupleEq<C,D>` override in the `Tuple<A,B>` class, one would first have to have establish 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♯ and GADT Haskell. In [?] we show that many interesting Haskell examples translate well, but some practically interesting ones, like Weihrich's type reconstruction algorithm taking untyped expressions to type expressions [?, ?], or the type safe LR parsers of Pottier and Régis-Gianas [?] do not. These all fail for reasons exemplified by our contrived, but small, `Eq` operation. Roughly speaking, C♯ can express all of the datatypes of GADT Haskell, but only some of its programs — which ones? C♯ 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♯ and, separately, into C♯ with equational constraints.

The structure of this paper is as follows. We start by presenting our object of study, a featherweight version of C♯, called C♯ 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♯'s typing of virtual methods and their overrides. We present a cast-free, type preserving translation from G minor to C♯ minor, proving that C♯ 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♯ and Java Generics, that has, surprisingly, gone unnoticed in the literature (we believe that [?] 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♯ major, an extension of C♯ 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 [?]). Like the system in [?], C♯ major both fixes the observed defect in C♯ minor, and extends the range of expressible GADT programs. We present a cast-free, type preserving translation from G major to C♯ major demonstrating that this variant of C♯ 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:

$$
\begin{tikzcd}
\text{G minor (§3)} \arrow[d, "(\text{§3.1})"] & \subset & \text{G major (§4)} \arrow[ll, dashed, bend right, "(\text{§6})"'] \arrow[lld, "(\text{§5.2})"'] \arrow[d, "(\text{§5.1})"] \\
\text{C}^\sharp\text{ minor (§2)} & \subset & \text{C}^\sharp\text{ major (§5)}
\end{tikzcd}
$$

2. C♯ minor

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

Syntax:

(class def)	cd	::=	$\texttt{class}\ C\texttt{<}\overline{X}\texttt{>}\ \texttt{:}\ I\ \texttt{where}\ \overline{E}\ \{\overline{T}\ \overline{f}\texttt{;}\ kd\ \overline{md}\}$
(constr def)	kd	::=	$\texttt{public}\ C\texttt{(}\overline{T}\ \overline{f}\texttt{)}\ \texttt{:}\ \texttt{base(}\overline{f}\texttt{)}\ \{\texttt{this.}\overline{f}\ \texttt{=}\ \overline{f}\texttt{;}\}$
(method qualifier)	Q	::=	$\texttt{public virtual}\ \mid\ \texttt{public override}$
(method def)	md	::=	$Q\ T\ m\texttt{<}\overline{X}\texttt{>(}\overline{T}\ \overline{x}\texttt{)}\ \texttt{where}\ \overline{E}\ \{\texttt{return}\ e\texttt{;}\}$
(expression)	e	::=	$x \mid e\texttt{.}f \mid e\texttt{.}m\texttt{<}\overline{T}\texttt{>(}\overline{e}\texttt{)} \mid \texttt{new}\ I\texttt{(}\overline{e}\texttt{)} \mid \texttt{(}T\texttt{)}e$
(value)	v, w	::=	$\texttt{new}\ I\texttt{(}\overline{v}\texttt{)}$
(type)	T, U, V	::=	$X \mid I$
(instantiated type)	I	::=	$C\texttt{<}\overline{T}\texttt{>}$
(equational constraint)	E	::=	$T\texttt{=}U$
(typing environment)	Γ	::=	$\overline{X}, \overline{x} : \overline{T}, \overline{E}$
(method signature)		::=	$\texttt{<}\overline{X}\ \texttt{where}\ \overline{E}\texttt{>}\overline{T} \rightarrow T \quad (\overline{X}$ is bound in $\overline{E}, \overline{T}, T)$

Type Equivalence:

$$\text{(eq-refl)}\ \frac{}{\Gamma \vdash X\texttt{=}X} \qquad \text{(eq-sym)}\ \frac{\Gamma \vdash U\texttt{=}T}{\Gamma \vdash T\texttt{=}U} \qquad \text{(eq-tran)}\ \frac{\Gamma \vdash T\texttt{=}U \quad \Gamma \vdash U\texttt{=}V}{\Gamma \vdash T\texttt{=}V}$$

$$\text{(eq-hyp)}\ \frac{T\texttt{=}U \in \Gamma}{\Gamma \vdash T\texttt{=}U} \qquad \text{(eq-con)}\ \frac{\Gamma \vdash \overline{T}\texttt{=}\overline{U}}{\Gamma \vdash C\texttt{<}\overline{T}\texttt{>=}C\texttt{<}\overline{U}\texttt{>}} \qquad \text{(eq-decon)}\ \frac{\Gamma \vdash C\texttt{<}\overline{T}\texttt{>=}C\texttt{<}\overline{U}\texttt{>}}{\Gamma \vdash T_i\texttt{=}U_i}$$

Subtyping:

$$\frac{\Gamma \vdash T\texttt{=}U}{\Gamma \vdash T <: U} \qquad \frac{\Gamma \vdash T <: U \quad \Gamma \vdash U <: V}{\Gamma \vdash T <: V} \qquad \frac{X \in \Gamma}{\Gamma \vdash X <: \texttt{object}}$$

$$\frac{\texttt{class}\ C\texttt{<}\overline{X}\texttt{>}\ \texttt{:}\ I\ \texttt{where}\ \overline{E}\ \{\ldots\}}{\Gamma \vdash C\texttt{<}\overline{T}\texttt{>} <: [\overline{T}/\overline{X}]I}$$

Well-formed contexts and types:

$$\frac{\overline{X} \vdash \overline{T}, \overline{U}, \overline{V}\ \mathit{ok}}{\vdash \overline{X}, \overline{x} : \overline{T}, \overline{U}\texttt{=}\overline{V}\ \mathit{ok}} \qquad \frac{X \in \Gamma}{\Gamma \vdash X\ \mathit{ok}}$$

$$\text{(ok-inst)}\ \frac{\texttt{class}\ C\texttt{<}\overline{X}\texttt{>}\ \texttt{:}\ I\ \texttt{where}\ \overline{E}\ \{\ldots\} \quad \Gamma \vdash \overline{T}\ \mathit{ok} \quad |\overline{T}| = |\overline{X}| \quad \Gamma \vdash [\overline{T}/\overline{X}]\overline{E}}{\Gamma \vdash C\texttt{<}\overline{T}\texttt{>}\ \mathit{ok}}$$

Typing:

$$\text{(ty-var)}\ \frac{}{\Gamma, x{:}T \vdash x : T} \qquad \text{(ty-fld)}\ \frac{\Gamma \vdash e : I \quad \mathit{fields}(I) = \overline{T}\ \overline{f}}{\Gamma \vdash e.f_i : T_i} \qquad \text{(ty-new)}\ \frac{\Gamma \vdash I\ \mathit{ok} \quad \mathit{fields}(I) = \overline{T}\ \overline{f} \quad \Gamma \vdash \overline{e} : \overline{T}}{\Gamma \vdash \texttt{new}\ I\texttt{(}\overline{e}\texttt{)} : I}$$

$$\text{(ty-cast)}\ \frac{\Gamma \vdash U\ \mathit{ok} \quad \Gamma \vdash e : T}{\Gamma \vdash \texttt{(}U\texttt{)}e : U} \qquad \text{(ty-sub)}\ \frac{\Gamma \vdash e : T \quad \Gamma \vdash T <: U \quad \Gamma \vdash U\ \mathit{ok}}{\Gamma \vdash e : U}$$

$$\text{(ty-meth)}\ \frac{\begin{array}{c}\Gamma \vdash e : I \quad \Gamma \vdash \overline{T}\ \mathit{ok} \quad \Gamma \vdash \overline{e} : [\overline{T}/\overline{X}]\overline{U} \\ \mathit{mtype}(I.m) = \texttt{<}\overline{X}\ \texttt{where}\ \overline{E}\texttt{>}\overline{U} \rightarrow U \quad \Gamma \vdash [\overline{T}/\overline{X}]\overline{E}\end{array}}{\Gamma \vdash e\texttt{.}m\texttt{<}\overline{T}\texttt{>(}\overline{e}\texttt{)} : [\overline{T}/\overline{X}]U}$$

Method and Class Typing:

$$\text{(ok-virtual)}\ \frac{\begin{array}{c}\texttt{class}\ C\texttt{<}\overline{X}\texttt{>}\ \texttt{:}\ I\ \texttt{where}\ \overline{E_1}\ \{\ldots\} \quad \mathit{mtype}(I.m)\ \text{not defined} \\ \overline{X}, \overline{Y} \vdash T, \overline{T}, \overline{E_2}\ \mathit{ok} \overline{X}, \overline{Y}, \overline{E_1}, \overline{E_2}, \overline{x}{:}\overline{T}, \texttt{this}{:}C\texttt{<}\overline{X}\texttt{>} \vdash e : T\end{array}}{\vdash \texttt{public virtual}\ T\ m\texttt{<}\overline{Y}\texttt{>(}\overline{T}\ \overline{x}\texttt{)}\ \texttt{where}\ \overline{E_2}\ \{\texttt{return}\ e\texttt{;}\}\ \mathit{ok\ in}\ C\texttt{<}\overline{X}\texttt{>}}$$

$$\text{(ok-override)}\ \frac{\begin{array}{c}\texttt{class}\ C\texttt{<}\overline{X}\texttt{>}\ \texttt{:}\ I\ \texttt{where}\ \overline{E_1}\ \{\ldots\} \quad \overline{X}, \overline{Y} \vdash T, \overline{T}\ \mathit{ok} \\ \mathit{mtype}(I.m) = \texttt{<}\overline{Y}\ \texttt{where}\ \overline{E_2}\texttt{>}\overline{T} \rightarrow T \\ \overline{X}, \overline{Y}, \overline{E_1}, \overline{E_2}, \overline{x}{:}\overline{T}, \texttt{this}{:}C\texttt{<}\overline{X}\texttt{>} \vdash e : T\end{array}}{\vdash \texttt{public override}\ T\ m\texttt{<}\overline{Y}\texttt{>(}\overline{T}\ \overline{x}\texttt{)}\ \{\texttt{return}\ e\texttt{;}\}\ \mathit{ok\ in}\ C\texttt{<}\overline{X}\texttt{>}}$$

$$\text{(ok-class)}\ \frac{\begin{array}{c}\overline{X} \vdash I, \overline{T}, \overline{E}\ \mathit{ok} \quad \mathit{fields}(I) = \overline{U}\ \overline{g} \quad \overline{f}\ \text{and}\ \overline{g}\ \text{disjoint} \\ \vdash \overline{md}\ \mathit{ok\ in}\ C\texttt{<}\overline{X}\texttt{>} \quad kd = \texttt{public}\ C\texttt{(}\overline{U}\ \overline{g}, \overline{T}\ \overline{f}\texttt{)}\ \texttt{base(}\overline{g}\texttt{)}\ \{\texttt{this.}\overline{f}\texttt{=}\overline{f}\texttt{;}\ \}\end{array}}{\vdash \texttt{class}\ C\texttt{<}\overline{X}\texttt{>}\ \texttt{:}\ I\ \texttt{where}\ \overline{E}\ \{\overline{T}\ \overline{f}\texttt{;}\ kd\ \overline{md}\}\ \mathit{ok}}$$

Figure 5. Syntax and typing rules for C$^\sharp$ minor (including highlighted changes for C$^\sharp$ major)

Evaluation rules:

$$\text{(e-fld)}\ \frac{e \Downarrow \texttt{new}\ I(\overline{v}) \quad \mathit{fields}(I) = \overline{T}\ \overline{f}}{e\,.f_i \Downarrow v_i} \qquad \text{(e-new)}\ \frac{\overline{e} \Downarrow \overline{v}}{\texttt{new}\ I(\overline{e}) \Downarrow \texttt{new}\ I(\overline{v})} \qquad \text{(e-cast)}\ \frac{e \Downarrow \texttt{new}\ I(\overline{v}) \quad \vdash I <: T}{(T)\,e \Downarrow \texttt{new}\ I(\overline{v})}$$

$$\text{(e-meth)}\ \frac{e \Downarrow \texttt{new}\ I(\overline{w}) \quad \mathit{mbody}(I.m\texttt{<}\overline{T}\texttt{>}) = \langle \overline{x}, e' \rangle \quad \overline{e} \Downarrow \overline{v} \quad [\overline{v}/\overline{x}, \texttt{new}\ I(\overline{w})/\texttt{this}]e' \Downarrow v}{e\,.m\texttt{<}\overline{T}\texttt{>(}\overline{e}\texttt{)} \Downarrow v}$$

Field lookup:

$$\frac{}{\mathit{fields}(\texttt{object}) = \{\}} \qquad \frac{\mathcal{D}(C) = \texttt{class}\ C\texttt{<}\overline{X}\texttt{>}\ \texttt{:}\ I\ \texttt{where}\ \overline{E}\ \{\overline{U_1}\ \overline{f_1}\texttt{;}\ kd\ \overline{md}\} \quad \mathit{fields}([\overline{T}/\overline{X}]I) = \overline{U_2}\ \overline{f_2}}{\mathit{fields}(C\texttt{<}\overline{T}\texttt{>}) = \overline{U_2}\ \overline{f_2}, [\overline{T}/\overline{X}]\overline{U_1}\ \overline{f_1}}$$

Method lookup:

$$\frac{\mathcal{D}(C) = \texttt{class}\ C\texttt{<}\overline{X_1}\texttt{>}\ \texttt{:}\ I\ \texttt{where}\ \overline{E}_1\ \{\ldots\ \overline{md}\} \quad m \text{ not defined } \texttt{public virtual} \text{ in } \overline{md}}{\mathit{mtype}(C\texttt{<}\overline{T_1}\texttt{>}.m) = \mathit{mtype}([\overline{T_1}/\overline{X_1}]I.m)}$$

$$\frac{\mathcal{D}(C) = \texttt{class}\ C\texttt{<}\overline{X_1}\texttt{>}\ \texttt{:}\ I\ \texttt{where}\ \overline{E}_1\ \{\ldots\ \overline{md}\} \quad \texttt{public virtual}\ U\ m\texttt{<}\overline{X_2}\texttt{>(}\overline{U}\ \overline{x}\texttt{)}\ \texttt{where}\ \overline{E}_2\ \{\texttt{return}\ e\texttt{;}\} \in \overline{md}}{\mathit{mtype}(C\texttt{<}\overline{T_1}\texttt{>}.m) = [\overline{T_1}/\overline{X_1}](\texttt{<}\overline{X_2}\ \texttt{where}\ \overline{E}_2\texttt{>}\overline{U} \to U)}$$

Method dispatch:

$$\frac{\texttt{class}\ C\texttt{<}\overline{X_1}\texttt{>}\ \texttt{:}\ I\ \{\ldots\ \overline{md}\} \quad m \text{ not defined in } \overline{md}}{\mathit{mbody}(C\texttt{<}\overline{T_1}\texttt{>}.m\texttt{<}\overline{T_2}\texttt{>}) = \mathit{mbody}([\overline{T_1}/\overline{X_1}]I.m\texttt{<}\overline{T_2}\texttt{>})}$$

$$\frac{\texttt{class}\ C\texttt{<}\overline{X_1}\texttt{>}\ \texttt{:}\ I\ \{\ldots\ \overline{md}\} \quad Q\ U\ m\texttt{<}\overline{X_2}\texttt{>(}\overline{U}\ \overline{x}\texttt{)}\ \{\texttt{return}\ e\texttt{;}\} \in \overline{md}}{\mathit{mbody}(C\texttt{<}\overline{T_1}\texttt{>}.m\texttt{<}\overline{T_2}\texttt{>}) = \langle \overline{x}, [\overline{T_1}/\overline{X_1}, \overline{T_2}/\overline{X_2}]e \rangle}$$

Figure 6. Evaluation rules and helper definitions for C$^\sharp$ minor (including highlighted changes for C$^\sharp$ major)

minor highlighted. For C$^\sharp$ 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$^\sharp$ minor really are valid C$^\sharp$ programs. The differences from Featherweight GJ are as follows:

- There are minor syntactic differences between Java and C$^\sharp$: 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 [**?**].

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 $C\texttt{<}\overline{T}\texttt{>}$ and ranged over by I; `object` abbreviates `object<>`.

A **class definition** cd consists of a class name C with formal type parameters $\overline{X}$, base class (superclass) I, constructor definition kd, typed instance fields $\overline{T}\ \overline{f}$ and methods $\overline{md}$. Method names in $\overline{md}$ 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 $\overline{X}$, formal argument names $\overline{x}$ and types $\overline{T}$, 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 $e\,.m\texttt{<}\overline{T}\texttt{>(}\overline{e}\texttt{)}$ or the creation of an object with initial field values $\texttt{new}\ I(\overline{e})$. A **value** v is a fully-evaluated expression, and (always) has the form $\texttt{new}\ I(\overline{v})$.

A **class table** $\mathcal{D}$ 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 $\Gamma = \overline{X}, \overline{x}{:}\overline{T}$ where free type variables in $\overline{T}$ are drawn from $\overline{X}$. We write $\cdot$ to denote the empty environment.

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

The operation $\mathit{mtype}(T.m)$, given a statically known class $T \equiv C\texttt{<}\overline{T}\texttt{>}$ 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 $\mathit{mbody}(T.m\texttt{<}\overline{T}\texttt{>})$, given a runtime class $T \equiv C\texttt{<}\overline{U}\texttt{>}$, method name m and method instantiation $\overline{T}$, walks the class hierarchy from C to find the most specific override of the virtual method, returning its body instantiated at types $\overline{T}$.

Theorem 1 (C$^\sharp$ minor evaluation preserves typing)**.** *Suppose $\mathcal{D}$ is valid and $\vdash^{\mathcal{D}} e : T$. If $e \Downarrow^{\mathcal{D}} v$ then $\vdash^{\mathcal{D}} v : T$.*

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$^\sharp$ minor. This demonstrated that generics in the style of C$^\sharp$ 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$^\sharp$ 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 $\Gamma = \overline{X}, \overline{x}{:}\overline{A}$, consists of sequences of type variable declarations, and type assignments to term variables. An enviroment is *valid* when $\overline{X}$ are distinct, $\overline{x}$ are distinct and all type variables free in $\overline{x}{:}\overline{A}$ are drawn from $\overline{X}$. A typing judgment $\Gamma \vdash M : A$ should be read "in the context of a typing environment Γ the term M has type A" with free type variables in M and A drawn from Γ. 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 $\mathcal{T}$, a function Σ mapping each type constructor $D \in \mathcal{T}$ to a finite set of term constructors $\mathcal{K}_D$, and each term constructor $k \in \mathcal{K}_D$ to a type of the form $\forall \overline{X_k}.(A_k \to D\,\overline{A_k})$. We assume that each type constructor D takes a fixed number $\mathrm{arity}(D) \geq 0$ of type arguments, that all constructor types are *closed*, and that the term constructors of distinct datatypes are disjoint, ie. $\mathcal{K}_D \cap \mathcal{K}_{D'} = \emptyset$, 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 Γ. 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 $k\,\overline{A}\,\overline{N}$ is typed as if k were a polymorphic function (compare the rule for function application). The elimination rule (case) is more subtle. The `case` term is annotated with a type function ϕ, of the same arity as the eliminated type constructor. The function, when instantiated at the type arguments of the scrutinee, determines the type of the entire case term. Each branch must be parametric in the type arguments to the term constructor, but the type of the branch varies according to the type arguments of the constructor's range.

To illustrate these features, consider the *Exp* type and *eval* function from Section 1, expressed in G minor:

$$\begin{array}{ll}\Sigma(\mathsf{Exp}) = \{ & \mathsf{Lit} : \mathsf{int} \to \mathsf{Exp}\,\mathsf{int} \\ & \mathsf{Plus} : \mathsf{Exp}\,\mathsf{int} \times \mathsf{Exp}\,\mathsf{int} \to \mathsf{Exp}\,\mathsf{int} \\ & \mathsf{Equals} : \mathsf{Exp}\,\mathsf{int} \times \mathsf{Exp}\,\mathsf{int} \to \mathsf{Exp}\,\mathsf{bool} \\ & \mathsf{Cond} : \forall Y.(\mathsf{Exp}\,\mathsf{bool} \times \mathsf{Exp}\ Y \times \mathsf{Exp}\ Y \to \mathsf{Exp}\ Y) \\ & \mathsf{Tuple} : \forall YZ.(\mathsf{Exp}\ Y \times \mathsf{Exp}\ Z \to \mathsf{Exp}\,(Y \times Z)) \\ & \mathsf{Fst} : \forall YZ.(\mathsf{Exp}\,(Y \times Z) \to \mathsf{Exp}\ Y)\ \} \end{array}$$

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

$$\begin{array}{l} \mathtt{rec}\ \mathsf{eval} = \Lambda X.\lambda(x{:}\mathsf{Exp}\ X){:}X. \\ \quad \mathtt{case}^{(X)X}\ x\ \mathtt{of} \\ \quad\quad \mathsf{Lit}\ y \Rightarrow y \\ \quad\quad \mathsf{Plus}\ y \Rightarrow \mathsf{eval}\,\mathsf{int}\,(\pi_1 y) + \mathsf{eval}\,\mathsf{int}\,(\pi_2 y) \\ \quad\quad \mathsf{Equals}\ y \Rightarrow \mathsf{eval}\,\mathsf{int}\,(\pi_1 y) = \mathsf{eval}\,\mathsf{int}\,(\pi_2 y) \\ \quad\quad \mathsf{Cond}\ Y\ y \Rightarrow \\ \quad\quad\quad \mathtt{if}\ \mathsf{eval}\,\mathsf{bool}\,(\pi_1 y)\ \mathtt{then}\ \mathsf{eval}\ Y\,(\pi_2 y)\ \mathtt{else}\ \mathsf{eval}\ Y\,(\pi_3 y) \\ \quad\quad \mathsf{Fst}\ YZ\ y \Rightarrow \pi_1(\mathsf{eval}\ Y \times Z\ y) \\ \quad\quad \mathsf{Tuple}\ YZ\ y \Rightarrow (\mathsf{eval}\ Y\,(\pi_1 y), \mathsf{eval}\ Z\,(\pi_2 y)) \end{array}$$

It 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$^\sharp$ minor

We now show how G minor programs can be translated to C$^\sharp$ minor, thus demonstrating that C$^\sharp$ 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$^\sharp$ minor type $A^\star$, together with global class definitions $\mathcal{G}$ used in the translation.

A polymorphic function type $\forall \overline{X}.(A \to B)$ is translated into a type-instantiation of a named function class, whose single polymorphic method `app<`$\overline{X}$`>` 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 `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 substitution on type parameters. This forces the translation of an open type such as $\forall X.(X \to Y)$ to be an instantiation of the *same* class as the translation of substitution instances such as $\forall X.(X \to \mathtt{int})$. 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 achive this we make use of an operation that Odersky and Läufer call "lifting" [?].

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

For example, the `X`-lifting of type `Fun<Fun<X,Y>,Fun<Y,Y>>` is the type abstraction $(\mathtt{Z1}, \mathtt{Z2})$`Fun<Fun<X,Z1>,Z2>` together with the types `Y` and `Fun<Y,Y>` which 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. *$A^\star = B^\star$ iff $A = B$.*

Syntax:

$$
\begin{array}{rlll}
\text{(types)} & A, B & ::= & X \mid \forall\overline{X}.(A \to B) \mid D\,\overline{A} \\
\text{(terms)} & M, N & ::= & x \mid M\,\overline{A}\,N \mid \texttt{rec}\ y = \Lambda\overline{X}.\lambda(x{:}A){:}B.M \mid k\,\overline{A}\,M \mid M@A \\
& & & \mid \texttt{case}^{\phi}\ M\ \texttt{of}\ \{k_k\,\overline{X_k}\,x_k \Rightarrow M_k\}_{k\in\mathcal{K}} \\
\text{(values)} & V, W & ::= & \texttt{rec}\ y = \Lambda\overline{X}.\lambda(x{:}A){:}B.M \mid k\,\overline{A}\,V \\
\text{(case contexts)} & \phi & ::= & (\overline{X})A \\
\text{(equations)} & \mathcal{E} & ::= & A \equiv B \\
\text{(typing env's)} & \Gamma & ::= & \overline{X},\ \overline{\mathcal{E}},\ \overline{x} : \overline{A}
\end{array}
$$

Well-formed contexts and types:

$$
\frac{\overline{X} \vdash \overline{A}, \overline{B}, \overline{C}\ \mathit{ok}}{\vdash \overline{X}, \overline{B \equiv C}, \overline{x}{:}\overline{A}\ \mathit{ok}}
\qquad
\frac{X \in \Gamma}{\Gamma \vdash X\ \mathit{ok}}
\qquad
\frac{\overline{X}, \Gamma \vdash A, B\ \mathit{ok}}{\Gamma \vdash \forall\overline{X}.(A \to B)\ \mathit{ok}}
\qquad
\frac{\Gamma \vdash \overline{A}\ \mathit{ok}}{\Gamma \vdash D\,\overline{A}\ \mathit{ok}}
$$

Equations:

$$
\text{(refl)}\,\frac{}{\Gamma, X \vdash X \equiv X}
\qquad
\text{(id)}\,\frac{A \equiv B \in \Gamma}{\Gamma \vdash A \equiv B}
\qquad
\text{(c1)}\,\frac{\Gamma \vdash \overline{A} \equiv \overline{B}}{\Gamma \vdash D\,\overline{A} \equiv D\,\overline{B}}
\qquad
\text{(c2)}\,\frac{\Gamma, \overline{X} \vdash A \equiv A' \quad \Gamma, \overline{X} \vdash B \equiv B'}{\Gamma \vdash \forall\overline{X}.(A \to B) \equiv \forall\overline{X}.(A' \to B')}
$$

$$
\text{(d1)}\,\frac{\Gamma, \overline{A} \equiv \overline{B} \vdash \mathcal{E}}{\Gamma, D\,\overline{A} \equiv D\,\overline{B} \vdash \mathcal{E}}
\qquad
\text{(d2)}\,\frac{\Gamma, \overline{X}, A \equiv B, A' \equiv B' \vdash \mathcal{E} \quad \vdash \Gamma, \mathcal{E}\ \mathit{ok}}{\Gamma, \forall\overline{X}.(A \to A') \equiv \forall\overline{X}.(B \to B') \vdash \mathcal{E}}
\qquad
\text{(sym)}\,\frac{\Gamma \vdash B \equiv A}{\Gamma \vdash A \equiv B}
\qquad
\text{(tran)}\,\frac{\Gamma \vdash A \equiv A' \quad \Gamma \vdash A' \equiv A''}{\Gamma \vdash A \equiv A''}
$$

Typing:

$$
\text{(var)}\,\frac{x{:}A \in \Gamma}{\Gamma \vdash x : A}
\qquad
\text{(abs)}\,\frac{\Gamma \vdash \forall\overline{X}.(A \to B)\ \mathit{ok} \qquad \Gamma, \overline{X}, x{:}A, y{:}\forall\overline{X}.(A \to B) \vdash M : B}{\Gamma \vdash \texttt{rec}\ y = \Lambda\overline{X}.\lambda(x{:}A){:}B.M : \forall\overline{X}.(A \to B)}
$$

$$
\text{(eqn)}\,\frac{\Gamma \vdash M : A \qquad \Gamma \vdash B\ \mathit{ok} \qquad \Gamma \vdash A \equiv B}{\Gamma \vdash M@B : B}
\qquad
\text{(app)}\,\frac{\Gamma \vdash M : \forall\overline{X}.(A \to B) \qquad \Gamma \vdash \overline{A}\ \mathit{ok} \qquad \Gamma \vdash N : [\overline{A}/\overline{X}]A}{\Gamma \vdash M\,\overline{A}\,N : [\overline{A}/\overline{X}]B}
$$

$$
\text{(inj)}\,\frac{\Sigma(D)(k) = k : \forall\overline{X}.A \to D\,\overline{B} \qquad \Gamma \vdash \overline{A}\ \mathit{ok} \qquad \Gamma \vdash N : [\overline{A}/\overline{X}]A}{\Gamma \vdash k\,\overline{A}\,N : D\,([\overline{A}/\overline{X}]\overline{B})}
$$

$$
\text{(eqn-case)}\,\frac{\begin{array}{c}\Sigma(D) = \{k : \forall\overline{X_k}.A_k \to D\,\overline{B_k}\}_{k\in\mathcal{K}} \quad \Gamma, \overline{X} \vdash C\ \mathit{ok} \qquad \Gamma \vdash M : D\,\overline{B} \\ \{\Gamma, \overline{X_k}, \overline{B \equiv B_k}, x_k{:}A_k \vdash M_k : [\overline{B_k}/\overline{X}]C\}_{k\in\mathcal{K}}\end{array}}{\Gamma \vdash \texttt{case}^{(\overline{X})C}\ M\ \texttt{of}\ \{k\,\overline{X_k}\,x_k \Rightarrow M_k\}_{k\in\mathcal{K}} : [\overline{B}/\overline{X}]C}
$$

Evaluation:

$$
\text{(e-val)}\,\frac{}{V \Downarrow V}
\qquad
\text{(e-eqn)}\,\frac{M \Downarrow V}{M@A \Downarrow V}
\qquad
\text{(e-app)}\,\frac{\begin{array}{c}M \Downarrow \texttt{rec}\ y = \Lambda\overline{X}.\lambda(x{:}A){:}B.M' \qquad N \Downarrow V \\ [V/x][\overline{A}/\overline{X}][\texttt{rec}\ y = \Lambda\overline{X}.\lambda(x{:}A){:}B.M'/y]M' \Downarrow W\end{array}}{M\,\overline{A}\,N \Downarrow W}
$$

$$
\text{(e-inj)}\,\frac{M \Downarrow V}{k\,\overline{A}\,M \Downarrow k\,\overline{A}\,V}
\qquad
\text{(e-case)}\,\frac{M \Downarrow k\,\overline{A}\,V \quad k \in \mathcal{K} \quad [V/x_k][\overline{A}/\overline{X_k}]M_k \Downarrow W}{\texttt{case}^{\phi}\ M\ \texttt{of}\ \{k\,\overline{X_k}\,x_k \Rightarrow M_k\}_{k\in\mathcal{K}} \Downarrow W}
$$

Figure 7. Syntax and semantics of G minor (and G major, highlighted additions)

Proof. Easy induction on structure of types, using the identification of `Fun` types up to renaming of type variables. □

Lemma 2. $([B/X]A)^\star = [B^\star/X]A^\star$.

Proof. Similar to [**?**]. □

Figure 8 defines the translation of terms. The translation of a term M is given by a judgment

$$\Gamma; \psi \vdash^{C} M : A \rightsquigarrow e \text{ in } \mathcal{D}$$

which should be read "In the context of typing environment Γ and argument environment ψ, term M with type A translates to an expression e and additional class definitions $\mathcal{D}$ using fresh class names prefixed by C".

When translating the body M of a function value $\texttt{rec}\ y = \Lambda\overline{X}.\lambda(x{:}A){:}B.M$ 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 Γ and argument environment ψ defined by the grammar

$$\psi ::= y(\overline{X}, x{:}A){:}B \mid k\ \overline{X}\ (x{:}A)$$

in which $y(\overline{X}, x{:}A){:}B$ denotes an environment used when translating functions in which $x{:}A$ is the argument and $y{:}\forall\overline{X}.(A \to B)$ is the function, and $k\ \overline{X}\ (x{:}A)$ denotes a constructor environment for constructor k in which $\overline{X}$ 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 [**?**].

The translation of function abstractions and case makes use of an operation $\Gamma \uplus \psi$ that pushes the bindings from ψ into Γ. It is

Global class definitions ($\mathcal{G}$):

For each $D \in \mathcal{T}$ define

$\texttt{class}\ D\texttt{<}\overline{X}\texttt{> : object}\ \{\ \}$

For each $k \in \mathcal{K}_D$ with $\Sigma(D)(k) = \forall \overline{X_k}.(A_k \to D\,\overline{A_k})$ define

$\texttt{class}\ Dk\texttt{<}\overline{X_k}\texttt{> :}\ D\texttt{<}\overline{A_k}^\star\texttt{>}\ \{\ A_k{}^\star\ x\texttt{; public}\ \ Dk\texttt{(}A_i{}^\star\ x\texttt{)}\ \{\ \texttt{this.}x\ \texttt{=}\ x\texttt{;}\}\ \}$

For all $\overline{X}$, T, U where $\mathrm{freevars}(T) \setminus \overline{X} = \overline{Y}$ and $\mathrm{freevars}(U) \setminus \overline{X} = \overline{Z}$ and identified up to renaming of $\overline{X\,Y\,Z}$ define

$\texttt{class Fun}_{(\overline{X})\,T \to U}\texttt{<}\overline{Y\,Z}\texttt{>}\ \{\ \texttt{public virtual}\ U\ \texttt{app<}\overline{X}\texttt{>(}T\ x\texttt{)}\{\ \texttt{return this.app<}\overline{X}\texttt{>(}x\texttt{);}\}\ \}$

Types:

$$
\begin{array}{rcl}
X^\star & = & X \\
(D\,\overline{A})^\star & = & D\texttt{<}\overline{A}^\star\texttt{>} \\
(\forall \overline{X}.(A \to B))^\star & = & \texttt{Fun}_{(\overline{X})\,T\to U}\texttt{<}\overline{T\,U}\texttt{>} \quad \text{where } \overline{X}\text{-lifting of } A^\star \text{ is } \langle (\overline{Y})\,T, \overline{T}\rangle \text{ and } \overline{X}\text{-lifting of } B^\star \text{ is } \langle (\overline{Z})\,U, \overline{U}\rangle
\end{array}
$$

Terms:

$$
\text{(tr-argvar)}\ \frac{}{\Gamma; y(\overline{X}, x{:}A){:}B \vdash^C x : A \leadsto x}
\qquad
\text{(tr-funvar)}\ \frac{}{\Gamma; y(\overline{X}, x{:}A){:}B \vdash^C y : \forall \overline{X}.(A \to B) \leadsto \texttt{this}}
$$

$$
\text{(tr-funfree)}\ \frac{}{\overline{X}, \overline{x}{:}\overline{A}; y(\overline{X}, x{:}A){:}B \vdash^C x_i : A_i \leadsto \texttt{this.}x_i}
\qquad
\text{(tr-casefree)}\ \frac{}{\overline{X}, \overline{x}{:}\overline{A}; k\ \overline{X}\ (x{:}A) \vdash^C x_i : A_i \leadsto x_i}
$$

$$
\text{(tr-casearg)}\ \frac{}{\Gamma; k\ \overline{X}\ (x{:}A) \vdash^C x : A \leadsto \texttt{this.}x}
\qquad
\text{(tr-inj)}\ \frac{\Sigma(D)(k) = \forall \overline{X}.(A \to B) \qquad \Gamma; \psi \vdash^C N : [\overline{A}/\overline{X}]A \leadsto e \text{ in } \mathcal{D}}{\Gamma; \psi \vdash^C k\,\overline{A}\,N : [\overline{A}/\overline{X}]B \leadsto \texttt{new}\ Dk\texttt{<}\overline{A}^\star\texttt{>(}e\texttt{)} \text{ in } \mathcal{D}}
$$

$$
\text{(tr-app)}\ \frac{\Gamma; \psi \vdash^{C1} M : \forall \overline{X}.(A \to B) \leadsto e \text{ in } \mathcal{D}_1 \qquad \Gamma; \psi \vdash^{C2} N : [\overline{A}/\overline{X}]A \leadsto e' \text{ in } \mathcal{D}_2}{\Gamma; \psi \vdash^C M\,\overline{A}\,N : [\overline{A}/\overline{X}]B \leadsto e\texttt{.app<}\overline{A^\star}\texttt{>(}e'\texttt{)} \text{ in } \mathcal{D}_1 \cup \mathcal{D}_2}
$$

$$
\text{(tr-abs)}\ \frac{\Gamma \uplus \psi; y(\overline{Y}, x{:}A){:}B \vdash^{C1} M : B \leadsto e \text{ in } \mathcal{D}_0 \quad \Gamma; \psi \vdash^C \overline{x} : \overline{A} \leadsto \overline{e}}{\Gamma; \psi \vdash^C \texttt{rec}\ y = \Lambda \overline{Y}.\lambda(x{:}A){:}B.M : \forall \overline{Y}.(A \to B) \leadsto \texttt{new}\ C\texttt{<}\overline{X}\texttt{>(}\overline{e}\texttt{)} \text{ in } \mathcal{D} \cup \mathcal{D}_0}
$$

$$
\Gamma \uplus \psi = \overline{X}, \overline{x}{:}\overline{A}
$$

$$
\mathcal{D} = \left\{ C \mapsto \begin{array}{l} \texttt{class}\ C\texttt{<}\overline{X}\texttt{> :}\ (\forall \overline{Y}.(A \to B))^\star \\ \{\ \overline{A}^\star\ \overline{x}\texttt{; public}\ \ C\texttt{(}\overline{A}^\star\ \overline{x}\texttt{)}\ \{\ \texttt{this.}\overline{x}\ \texttt{=}\ \overline{x}\texttt{;}\} \\ \quad \texttt{public override}\ B^\star\ \texttt{app<}\overline{Y}\texttt{>(}A^\star\ x\texttt{)}\ \{\ \texttt{return}\ e\texttt{;}\ \}\ \} \end{array} \right\}
$$

$$
\text{(tr-case)}\ \frac{\begin{array}{c} \Sigma(D) = \{k : \forall \overline{X_k}.A_k \to D\,\overline{B_k}\}_{k \in \mathcal{K}} \qquad \Gamma; \psi \vdash^{C0} M : D\,\overline{B} \leadsto e \text{ in } \mathcal{D}_0 \\ \Gamma; \psi \vdash^C \overline{x} : \overline{A} \leadsto \overline{e} \qquad \{\Gamma \uplus \psi; k\,\overline{X_k}\ (x_k{:}A_k) \vdash^{Ck} M_k : [\overline{B_k}/\overline{Y}]B \leadsto e_k \text{ in } \mathcal{D}_k\}_{k \in \mathcal{K}_D} \end{array}}{\Gamma; \psi \vdash^C \texttt{case}^{(\overline{Y})B}\ M\ \texttt{of}\ \{k\,\overline{X_k}\,x_k \Rightarrow M_k\}_{k \in \mathcal{K}} : [\overline{B}/\overline{Y}]B \leadsto e\texttt{.case}C\texttt{<}\overline{X}\texttt{>(}\overline{e}\texttt{)} \text{ in } \mathcal{D} \cup \mathcal{D}_0 \cup \bigcup_{k \in \mathcal{K}} \mathcal{D}_k}
$$

$$
\Gamma \uplus \psi = \overline{X}, \overline{x}{:}\overline{A}
$$

$$
\mathcal{D} = \left\{ \begin{array}{ll} D \mapsto & \texttt{class}\ D\texttt{<}\overline{Y}\texttt{> : object}\ \{\ \ \texttt{public virtual}\ B^\star\ \texttt{case}C\texttt{<}\overline{X}\texttt{>(}\overline{A}^\star\ \overline{x}\texttt{)}\ \{\ \ \texttt{return this.case}C\texttt{<}\overline{X}\texttt{>(}\overline{x}\texttt{);}\}\}, \\ Dk \mapsto & \texttt{class}\ Dk\texttt{<}\overline{X_k}\texttt{> :}\ (D\,\overline{B_k})^\star\ \{\ \ \texttt{public override}\ ([\overline{B_k}/\overline{Y}]B)^\star\ \texttt{case}C\texttt{<}\overline{X}\texttt{>(}\overline{A}^\star\ \overline{x}\texttt{)}\{\ \ \texttt{return}\ e_k\texttt{;}\ \}\} \end{array} \right\}
$$

Figure 8. Translation of types and terms

defined as follows:

$$
\begin{array}{rcl}
\Gamma \uplus y(\overline{X}, x{:}A){:}B & = & \overline{X}, \Gamma, x{:}A, y : \forall \overline{X}.(A \to B) \\
\Gamma \uplus k\ \overline{X}\ (x{:}A) & = & \overline{X}, \Gamma, x{:}A
\end{array}
$$

The translation is essentially defined by induction over the structure of the typing derivation of a term: $\Gamma; \psi \vdash^C M : A \leadsto e$ in $\mathcal{D}$ is defined when $\Gamma \uplus \psi \vdash M : A$.

Consider rules (tr-inj) and (tr-case) for GADT introduction and elimination. Datatype constructors are translated to a simple use of `new` on the appropriate constructor class. The `case` construct is translated to a `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* $\vdash^{\mathit{min}} M : A \leadsto e$ in $\mathcal{D}$ *then* $\mathcal{D} \cup \mathcal{G}$ *is a valid class table and* $\vdash^{\mathcal{D} \cup \mathcal{G}} e : A^\star$.

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 $\overline{B}$ 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:

$$\begin{array}{l}
\Sigma(\mathsf{List}) = \{\mathsf{Nil} : \forall Y.(\mathsf{unit} \to \mathsf{List}\ Y) \\
\qquad\qquad\quad \mathsf{Cons} : \forall Y.(Y \times \mathsf{List}\ Y \to \mathsf{List}\ Y)\} \\
\mathtt{rec}\ \mathsf{sum} = \lambda(x{:}\mathsf{List\,int}){:}\mathsf{int}.\mathsf{case}^{\phi}\ x\ \mathtt{of} \\
\qquad \mathsf{Nil}\ Y\ y \Rightarrow 0 \\
\qquad \mathsf{Cons}\ Y\ y \Rightarrow (\pi_1 y) + \mathsf{sum}\ (\pi_2 y)
\end{array}$$

Although (a variant of) this is legal in G major, the function is untypable in G minor: in the second branch, the type of $\pi_1 y$ is just Y, not int, so we can't pass it to +; the recursive call is illegal because $\pi_1 y$ has type List Y, not List int. Luckily, every instantiation-specific case expression over a PADT such as List can be translated into an application of a polymorphic, higher-order function, match, that, given an instantiation $\overline{X}$, result type R and a value of the PADT, returns a higher-order function that selects and applies the appropriate continuation from its suite of arguments. For List X we can define, first match, then sum as follows:

$$\begin{array}{l}
\mathtt{rec}\ \mathsf{match} \\
= \Lambda XR.\lambda(x{:}\mathsf{List}\,X){:}(\mathsf{unit} \to R) \to (X \times \mathsf{List}\ X \to R) \to R. \\
\qquad \mathsf{case}^{(Z)(\mathsf{unit}\to R)\to(Z\times\mathsf{List}\ Z\to R)\to R}\ x\ \mathtt{of} \\
\qquad\quad \mathsf{Nil}\ Y\ y \Rightarrow \lambda f{:}\mathsf{unit} \to R.\lambda g{:}Y \times \mathsf{List}\ Y \to R.f\ y \\
\qquad\quad \mathsf{Cons}\ Y\ y \Rightarrow \lambda f{:}\mathsf{unit} \to R.\lambda g{:}Y \times \mathsf{List}\ Y \to R.g\ y \\
\mathtt{rec}\ \mathsf{sum} = \lambda(x{:}\mathsf{List\,int}){:}\mathsf{int}.\mathsf{match}\ \ \mathsf{int}\ \mathsf{int}\ \ x \\
\qquad (\lambda y{:}\mathsf{unit}.0) \\
\qquad (\lambda y{:}\mathsf{int} \times \mathsf{List\,int}.(\pi_1 y) + \mathsf{sum}\ (\pi_2 y))
\end{array}$$

Cleary, rule (case), though expressive enough to capture some GADT programs, is rather feeble at coping with workaday PADTs, requiring higher-order chores for first-order tasks. A functional programmer would find this tedious, but, surprisingly, that's precisely what the object-oriented programmer puts up with using Generics. For instance, to implement the summation function into C$^\sharp$, one basically has two alternatives (Figure 9). The first is to implement an unsafe virtual method `UnsafeSum` that uses runtime casts to recover that `T` must actually be `int`. Because this approach relies on exact runtime types, it is not an option in G minor and cannot be checked at runtime in Java (due to its erasure semantics). The second is to introduce a safe, generic visitor interface, `IVisit<T,R>`, that encapsulates a suite of methods, and visit the list from a static method `AwkwardSum`, passing an instance of a specific visitor class as the (higher-order) method suite. `AwkwardSum` is closely related to the G minor workaround discussed above: method `Accept` corresponds to the higher-order function match, its `IVisit<T,R>` argument to the (uncurried) suite of continuations f, g. Unfortunately, in the object-oriented setting this workaround is even more distasteful: introducing the visitor pattern fixes the class hierarchy rooted a `List<T>` — it is no longer extensible by future subclasses since adding a new subclass requires a modification to the `IVisit` interface. To implement `UnsafeSum` safely, we just need to capture the invariant that it will only be invoked when `T` is `int`: this is expressed by the constrained declaration of `SafeSum`, which uses our proposed extension [?] and is both natural and cast-free.

Why can't we apply the higher-order workaround to those problematic GADT programs that require type specific case? The problem with GADTs is that defining a useful visitor interface itself requires placing equational constraints on the visitor methods [?].

```
public abstract class List<T> {
  public abstract int UnsafeSum();
  public abstract R Accept<R>(IVisit<T,R> v);
  public static int AwkwardSum(List<int> l)
    { return l.Accept<int>(new SumVisitor()); }
  public abstract int SafeSum() where T = int;
}
public class Nil<A> : List<A> {
  public override int UnsafeSum(){ return 0;}
  public override R Accept<R>(IVisit<A,R> v)
    { return v.VisitNil(this);}
  public override int SafeSum() // where A = int
    { return 0;}
}
public class Cons<A> : List<A> {
  public A head; public List<A> tail;
  public override int UnsafeSum()
   { return ((int) (object) this.head)
             + this.tail.UnsafeSum();}
  public override R Accept<R>(IVisit<A,R> v)
   { return v.VisitCons(this);}
  public override int SafeSum() // where A = int
   { return this.head + this.tail.SafeSum();}
}
public interface IVisit<T, R> {
  R VisitNil(Nil<T> n); R VisitCons(Cons<T> c);
}
public class SumVisitor: IVisit<int,int> {
  public int VisitNil(Nil<int> that){ return 0;}
  public int VisitCons(Cons<int> that)
    { return that.head + that.tail.Accept(this);}
}
```

Figure 9. Summing list of integers

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 retype M at a derivably 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 [?] 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 $\overline{B_k}$ of the type of the pattern with the actual instantiation $\overline{B}$ of the type of the scrutinee, potentially inducing more equations on both the type variables bound by the pattern $\overline{X_k}$ and the ambient type variables in Γ.

When we wish to be explicit, we write $\Gamma \vdash^{\text{min}} M : A$ for typing judgments in G minor, and $\Gamma \vdash^{\text{maj}} M : A$ for G major.

```
rec sum = λ(x:List int):int.case^(X)int x of
    Nil Y y ⇒ 0
    Cons Y y ⇒ ((π₁y)@int) + sum ((π₂y)@List int)    // Y ≡ int

rec eq = ΛX.λ(x:Exp X × Exp X):bool.
  case^(X)bool π₁x of
    Lit y ⇒ case π₂x of
          Lit z ⇒ y = z
          ... ⇒ false
    Tuple YY′ y ⇒    // X ≡ Y × Y′
      case π₂x of
        Tuple ZZ′ z ⇒    // X ≡ Z × Z′
          eq Y (π₁y, π₁z@Exp Y) ∧ eq Y′ (π₂y, π₂z@Exp Y′)
        ... ⇒ false
```

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

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

Figure 10 presents the problematic sum and Eq functions from Sections 3.2 and 1, both written in G major. Notice the @ 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* $\Gamma \vdash^{min} M : A$ *then* $\Gamma \vdash^{maj} M : A$.

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 $(\overline{X})C$. When this type function is constant (*i.e.* $\overline{X}$ are not free in C) then refinement only occurs through equations. The following lemma shows that equations alone suffice.

Lemma 4. *If* $\Gamma \vdash^{maj} M : A$ *then there is a term* N *such that* $\Gamma \vdash^{maj} N : A$*, 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

$$\texttt{case}^{(\overline{X})C}\ M\ \texttt{of}\ \{k\,\overline{X_k}\,x_k \Rightarrow M_k\}_{k\in\mathcal{K}}$$

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

$$\texttt{case}^{(\overline{Z})C'}\ M'\ \texttt{of}\ \{k\,\overline{X_k}\,x_k \Rightarrow M'_k@C'\}_{k\in\mathcal{K}}$$

which is well-typed because branch body M'_k of type $[\overline{B_k}/\overline{X}]C$ can be re-typed at C' using rule (eqn) and the equation $\overline{B} \equiv \overline{B_k}$ from the context. □

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

Proof. Induction on the derivation of $\mathcal{J}$. □

Theorem 4 (Evaluation preserves typing)**.** *If* $\vdash M : A$ *and* $M \Downarrow V$ *then* $\vdash V : A$.

Proof. Induction on the evaluation derivation. □

5. Adding equations to C$^\sharp$

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$^\sharp$ minor, called C$^\sharp$ major. For a more gentle exposition, see [?]. The syntax, typing and helper definitions of C$^\sharp$ major are shown in Figures 5 and 6, but this time including the highlighted bits. In brief, C$^\sharp$ major extends C$^\sharp$ 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 *mtype*, 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$^\sharp$ 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$^\sharp$ 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$^\sharp$ major is the following lemma, that allows one to discharge established equational hypotheses from typing judgments:

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

Proof. Induction on the derivation of $\mathcal{J}$. □

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

Theorem 5 (C$^\sharp$ major evaluation preserves typing)**.** *Suppose that* $\mathcal{D}$ *is a valid class table and* $\vdash^{\mathcal{D}} e : T$*. If* $e \Downarrow^{\mathcal{D}} v$ *then* $\vdash^{\mathcal{D}} v : T$.

Constructor environment:

$$\psi ::= y(\overline{X}, x{:}A){:}B \;\mid\; k\ \overline{X}, \overline{\mathcal{E}}\ (x{:}A)$$

$$\begin{array}{rcl} \Gamma \uplus y(\overline{X}, x{:}A){:}B & = & \overline{X}, \Gamma, x{:}A, y : \forall\overline{X}.(A \to B) \\ \Gamma \uplus k\ \overline{X}, \overline{\mathcal{E}}\ (x{:}A) & = & \overline{X}, \Gamma, \overline{\mathcal{E}}, x{:}A \end{array}$$

Translation of terms:

$$\text{(tr-argvar)}\ \frac{}{\Gamma; y(\overline{X}, x{:}A){:}B \vdash^{C} x : A \rightsquigarrow x} \qquad \text{(tr-funvar)}\ \frac{}{\Gamma; y(\overline{X}, x{:}A){:}B \vdash^{C} y : \forall\overline{X}.(A \to B) \rightsquigarrow \texttt{this}}$$

$$\text{(tr-funfree)}\ \frac{}{\overline{X}, \overline{x{:}A}; y(\overline{X}, x{:}A){:}B \vdash^{C} x_i : A_i \rightsquigarrow \texttt{this.}x_i} \qquad \text{(tr-casefree)}\frac{}{\overline{X}, \overline{x{:}A}; k\ \overline{X}, \overline{\mathcal{E}}\ (x{:}A) \vdash^{C} x_i : A_i \rightsquigarrow x_i}$$

$$\text{(tr-casearg)}\frac{}{\Gamma; k\ \overline{X}, \overline{\mathcal{E}}\ (x{:}A) \vdash^{C} x : A \rightsquigarrow \texttt{this.}x} \qquad \text{(tr-inj)}\ \frac{\Sigma(D)(k) = \forall\overline{X}.(A \to B) \qquad \Gamma; \psi \vdash^{C} N : [\overline{A}/\overline{X}]A \rightsquigarrow e \text{ in } \mathcal{D}}{\Gamma; \psi \vdash^{C} k\,\overline{A}\,N : [\overline{A}/\overline{X}]B \rightsquigarrow \texttt{new } Dk\texttt{<}\overline{A}^{\star}\texttt{>(}e\texttt{)} \text{ in } \mathcal{D}}$$

$$\text{(tr-app)}\ \frac{\Gamma; \psi \vdash^{C1} M : \forall\overline{X}.(A \to B) \rightsquigarrow e \text{ in } \mathcal{D}_1 \qquad \Gamma; \psi \vdash^{C2} N : [\overline{A}/\overline{X}]A \rightsquigarrow e' \text{ in } \mathcal{D}_2}{\Gamma; \psi \vdash^{C} M\,\overline{A}\,N : [\overline{A}/\overline{X}]B \rightsquigarrow e\texttt{.app<}\overline{A^{\star}}\texttt{>(}e'\texttt{)} \text{ in } \mathcal{D}_1 \cup \mathcal{D}_2}$$

$$\text{(tr-eqn)}\ \frac{\Gamma; \psi \vdash^{C} M : A \rightsquigarrow e \text{ in } \mathcal{D} \quad \Gamma \uplus \psi \vdash A \equiv B}{\Gamma; \psi \vdash^{C} M@B : B \rightsquigarrow e \text{ in } \mathcal{D}}$$

$$\text{(tr-abs)}\ \frac{\Gamma \uplus \psi; y(\overline{Y}, x{:}A){:}B \vdash^{C1} M : B \rightsquigarrow e \text{ in } \mathcal{D}_0 \quad \Gamma; \psi \vdash^{C} \overline{x} : \overline{A} \rightsquigarrow \overline{e}}{\Gamma; \psi \vdash^{C} \texttt{rec } y = \Lambda\overline{Y}.\lambda(x{:}A){:}B.M : \forall\overline{Y}.(A \to B) \rightsquigarrow \texttt{new } C\texttt{<}\overline{X}\texttt{>(}\overline{e}\texttt{)} \text{ in } \mathcal{D} \cup \mathcal{D}_0}$$

$$\Gamma \uplus \psi = \overline{X}, \overline{\mathcal{E}}, \overline{x{:}A}$$

$$\mathcal{D} = \left\{ C \mapsto \begin{array}{l} \texttt{class } C\texttt{<}\overline{X}\texttt{> : } (\forall\overline{Y}.(A \to B))^{\star}\ \texttt{where } \overline{\mathcal{E}}^{\star} \\ \{\ \overline{A}^{\star}\ \overline{x}\texttt{; public } C\texttt{(}\overline{A}^{\star}\ \overline{x}\texttt{)}\ \{\ \texttt{this.}\overline{x}\texttt{ = }\overline{x}\texttt{;}\} \\ \quad \texttt{public override } B^{\star}\ \texttt{app<}\overline{Y}\texttt{>(}A^{\star}\ x\texttt{)}\ \{\ \texttt{return } e\texttt{;}\ \}\ \} \end{array} \right\}$$

$$\text{(tr-eqn-case)}\ \frac{\begin{array}{l} \Sigma(D) = \{k : \forall\overline{X_k}.A_k \to D\,\overline{B_k}\}_{k\in\mathcal{K}} \qquad \Gamma; \psi \vdash^{C0} M : D\,\overline{B} \rightsquigarrow e \text{ in } \mathcal{D}_0 \\ \Gamma; \psi \vdash^{C} \overline{x} : \overline{A} \rightsquigarrow \overline{e} \qquad \{\Gamma \uplus \psi; k\ \overline{X_k}, \overline{B} \equiv \overline{B_k}\ (x_k{:}A_k) \vdash^{Ck} M_k : [\overline{B_k}/\overline{Y}]B \rightsquigarrow e_k \text{ in } \mathcal{D}_k\}_{k\in\mathcal{K}} \end{array}}{\Gamma; \psi \vdash^{C} \texttt{case}^{(\overline{Y})B}\ M\ \texttt{of}\ \{k\,\overline{X_k}\,x_k \Rightarrow M_k\}_{k\in\mathcal{K}} : [\overline{B}/\overline{Y}]B \rightsquigarrow e\texttt{.case}C\texttt{<}\overline{X}\texttt{>(}\overline{e}\texttt{)} \text{ in } \mathcal{D} \cup \mathcal{D}_0 \cup \bigcup_{k\in\mathcal{K}} \mathcal{D}_k}$$

$$\Gamma \uplus \psi = \overline{X}, \overline{\mathcal{E}}, \overline{x{:}A}$$

$$\mathcal{D} = \left\{ \begin{array}{ll} D \mapsto & \texttt{class } D\texttt{<}\overline{Y}\texttt{>}\ \{\ \texttt{public virtual } B^{\star}\ \texttt{case}C\texttt{<}\overline{X}\texttt{>(}\overline{A}^{\star}\ \overline{x}\texttt{)}\ \texttt{where } \overline{\mathcal{E}}^{\star}, \overline{Y}\texttt{=}\overline{B}^{\star}\ \{\ \texttt{return this.case}C\texttt{<}\overline{X}\texttt{>(}\overline{x}\texttt{);}\}\ \}, \\ Dk \mapsto & \texttt{class } Dk\texttt{<}\overline{X_k}\texttt{> : } (D\,\overline{B_k})^{\star}\ \{\ \texttt{public override } ([\overline{B_k}/\overline{Y}]B)^{\star}\ \texttt{case}C\texttt{<}\overline{X}\texttt{>(}\overline{A}^{\star}\ \overline{x}\texttt{)}\ \{\texttt{return } e_k\texttt{;}\ \}\ \} \end{array} \right\}$$

Figure 11. Translation from G major to C$^\sharp$ major

5.1 Translation from G major to C$^\sharp$ major

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

The argument environment ψ is extended with equations guarding a constructor; these equations are propagated into the context Γ through the $\uplus$ operation. The (tr-abs) rules closes over equations by declaring them in the closure class; analogously, the (tr-eqn-case) rule closes over equations by declaring them on the `case` method. In addition, (tr-eqn-case) declares an equation $\overline{Y}\texttt{=}\overline{B}^{\star}$ which gets refined in constructor subclasses to $\overline{B_k}^{\star}\texttt{=}\overline{B}^{\star}$ as we require.

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

Proof. By induction on the equality derivation. ◻

Lemma 8. *If $\Gamma \vdash A \equiv B$ then $\Gamma^{\star} \vdash A^{\star}\texttt{=}B^{\star}$.*

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

Theorem 6 (Translation preserves types)**.** *If $\vdash^{maj} M : A \rightsquigarrow e$ in $\mathcal{D}$ then $\mathcal{D} \cup \mathcal{G}$ is a valid class table and $\vdash^{\mathcal{D}\cup\mathcal{G}} e : A^{\star}$.*

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

5.2 Translation from G major to C$^\sharp$ minor

C$^\sharp$ 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-eqn) rule to be:

$$\text{(tr-eqn)}\ \frac{\Gamma; \psi \vdash^{C} M : A \rightsquigarrow e \text{ in } \mathcal{D} \quad \Gamma \uplus \psi \vdash A \equiv B}{\Gamma; \psi \vdash^{C} M@B : B \rightsquigarrow \texttt{(}B^{\star}\texttt{)}e \text{ in } \mathcal{D}}$$

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$^\sharp$ 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$^\sharp$ 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:

$$\Sigma(D) = \{\ k_1 : \forall X.(X \to D\ X),\ k_2 : \mathsf{bool} \to D\ \mathsf{int}\ \}$$
$$\lambda x{:}D\ \mathsf{int}. \left(\begin{array}{l} \mathtt{case}^{(Y)\mathsf{bool}}\ x\ \mathtt{of} \\ \quad k_1\ X\ y \Rightarrow y@\mathsf{int} = 5 \quad //X \equiv \mathsf{int} \\ \quad k_2\ z \Rightarrow z \end{array}\right)$$

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

$$\lambda x{:}D\ \mathsf{int}. \left(\begin{array}{l} \mathtt{case}^{(Y)(Y\to\mathsf{int})\to\mathsf{bool}}\ x\ \mathtt{of} \\ \quad k_1\ X\ y \Rightarrow \lambda f{:}(X \to \mathsf{int}).f\ y = 5 \\ \quad k_2\ z \Rightarrow \lambda f{:}(\mathsf{int} \to \mathsf{int}).z \end{array}\right) (\lambda w{:}\mathsf{int}.w)$$

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$^\sharp$ 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$^\sharp$ minor (hence C$^\sharp$) is at least 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 [**?**]. In future, it would be nice to *prove* that there are G major (cast-free C$^\sharp$ major) programs, such as *eq* (`Eq`), that cannot be expressed in G minor (cast-free C$^\sharp$ minor), pinning down the expressivity of the weaker systems, but that would require more sophisticated methods.

We originally believed that C$^\sharp$ 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$^\sharp$. 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$^\sharp$ 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$^\sharp$ and Java Generics. It seems odd for Generics to provide elegant support for some, but not all, GADT programs, but rather poor support for programming with ordinary PADTs such as `List<T>`. Of course, one could also argue that instantiation specific operations, such as `Sum`, 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 extensibilty, 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$^\sharp$ major. The observation that, without constraints, GADT's do not admit Visitor patterns [**?**], also hints at an incompleteness in Generics that is remedied by our extension.

References

[1] J. Cheney and R. Hinze. First-class phantom types. Technical Report 1901, Cornell University, 2003.

[2] A. Hejlsberg, S. Wiltamuth, and P. Golde. C# version 2.0 specification, 2005. Available from `http://msdn.microsoft.com/vcsharp/team/language/default.aspx`.

[3] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 1999.

[4] A. J. Kennedy and C. Russo. Generalized algebraic data types and object-oriented programming. In *OOPSLA '05: 20th annual ACM SIGPLAN conference on Object-oriented Programming Systems, Languages, and Applications*. ACM Press, 2005.

[5] A. J. Kennedy and D. Syme. Design and implementation of generics for the .NET Common Language Runtime. In *Programming Language Design and Implementation*. ACM, 2001.

[6] A. J. Kennedy and D. Syme. Transposing F to C$^\sharp$: Expressivity of parametric polymorphism in an object-oriented language. *Concurrency and Computation: Practice and Experience*, 16:707–733, 2004.

[7] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *ACM Symposium on Principles of Programming Languages*, pages 271–283, St. Petersburg, FL, January 1996.

[8] M. Odersky and K. Läufer. Putting type annotations to work. In *ACM Symposium on Principles of Programming Languages*. ACM, 1996.

[9] S. Peyton Jones, G. Washburn, and S. Weirich. Wobbly types: type inference for generalised algebraic data types. Draft, July 2004.

[10] Simon Peyton Jones et al. The ghc compiler version 6.4, March 2005. Download at `http://haskell.org/ghc`.

[11] F. Pottier and N. Gauthier. Polymorphic typed defunctionalization. In *31st ACM Symposium on Principles of Programming Languages (POPL'04)*, pages 89–98, Venice, Italy, January 2004.

[12] F. Pottier and Y. Régis-Gianas. Towards efficient, typed LR parsers. In *ACM Workshop on ML*, Electronic Notes in Theoretical Computer Science, pages 149–173, September 2005.

[13] F. Pottier and Y. Régis-Gianas. Stratified type inference for generalized algebraic data types. In *Proceedings of the 33st ACM Symposium on Principles of Programming Languages (POPL'06)*, Charleston, South Carolina, January 2006.

[14] Stephanie Weirich. Type-checker to generate typed term from untyped source, September 2004. Response to challenge at Dagstuhl'04 set by Lennart Augustsson. In ghc regression suite (`tc.hs`).

[15] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 224–235. ACM Press, 2003.