Generalized Algebraic Data Types and Object-Oriented Programming

Andrew Kennedy
akenn@microsoft.com

Claudio V. Russo
crusso@microsoft.com

Microsoft Research Ltd, 7JJ Thomson Ave, Cambridge, United Kingdom

ABSTRACT

Generalized algebraic data types (GADTs) have received much attention recently in the functional programming community. They generalize the (type) parameterized algebraic datatypes (PADTs) of ML and Haskell by permitting value constructors to return specific, rather than parametric, type-instantiations of their own datatype. GADTs have a number of applications, including strongly-typed evaluators, generic pretty-printing, generic traversals and queries, and typed LR parsing. We show that existing object-oriented programming languages such as Java and C♯ can express GADT definitions, and a large class of GADT-manipulating programs, through the use of generics, subclassing, and virtual dispatch. However, some programs can be written only through the use of redundant runtime casts. Moreover, instantiation-specific, yet safe, operations on ordinary PADTs only admit indirect cast-free implementations, via higher-order encodings. We propose a generalization of the type constraint mechanisms of C♯ and Java to both avoid the need for casts in GADT programs and higher-order contortions in PADT programs; we present a Visitor pattern for GADTs, and describe a refined `switch` construct as an alternative to virtual dispatch on datatypes. We formalize both extensions and prove type soundness.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features— *constraints, data types and structures, polymorphism, classes and objects, inheritance*; F.3.3 [**Logic and Meanings of Programs**]: Studies of Program Constructs—*type structure, object-oriented constructs*

General Terms

Languages, Theory

Keywords

Generalized algebraic data types, generics, constraints

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
OOPSLA'05, October 16–20, 2005, San Diego, California, USA.
Copyright 2005 ACM 1-59593-031-0/05/0010 ...$5.00.

1. INTRODUCTION

Consider implementing a little language using an object-oriented programming language such as Java or C♯. Abstract syntax trees in the language would typically be represented using an abstract class of expressions, with a concrete subclass for each node type. An interpreter for the language can be implemented by an abstract 'evaluator' method in the expression class, overridden for each node type. This is an instance of the *Interpreter* design pattern [6].

For example, take a language of integer, boolean and binary tuple expressions:

$$\begin{array}{rcl} exp & ::= & con \mid exp \texttt{ + } exp \mid exp \texttt{ - } exp \mid exp \texttt{ == } exp \\ & & \mid exp \texttt{ \&\& } exp \mid exp \texttt{ || } exp \mid exp \texttt{ ? } exp \texttt{ : } exp \\ & & \mid (exp, exp) \mid \texttt{fst}(exp) \mid \texttt{snd}(exp) \end{array}$$

C♯ code to implement this abstract syntax and its interpreter is shown in Figure 1. Note in particular two points. First, the result of the `eval` method has the universal type `object`, as expressions can evaluate to integers, booleans or pairs. Second, evaluation can fail due to type errors: adding an integer to a boolean throws `InvalidCastException`.

Now suppose that we decide to add static type-checking to the language, for example checking that arithmetic operations are applied only to integer expressions and that conditional expressions take a boolean expression as condition and two expressions of the same type for the branches. We could easily add a method that checks the type of an expression. This would then assure us that evaluation cannot fail with a type error; however, the runtime casts in the evaluator code (*e.g.* `(int)` in `Plus.Eval`) are still necessary to convince C♯ of the safety of the evaluator.

Now consider building types into the AST representation itself, using the *generics* feature recently added to C♯ and Java to parameterize the `Exp` class by the type of expressions that it represents. Then we can:

- define `Exp<T>` and its subclasses to represent expressions of type `T` that are *type correct by construction*;
- give `Eval` the result type `T` and *guarantee absence of type errors during evaluation.*

Figure 2 lists C♯ code that does just this. Observe how the type parameter of `Exp` is refined in subclasses; moreover, this refinement 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 the dynamically-

```
namespace U { // Untyped expressions
  public class Pair {
    public object fst, snd;
    public Pair(object fst, object snd)
      { this.fst=fst; this.snd=snd; }
  }
  public abstract class Exp
    { public abstract object Eval(); }
  public class Lit : Exp {
    int value;
    public Lit(int value) { this.value=value; }
    public override object Eval() { return value; }
  }
  public class Plus : Exp { // Likewise for Minus, etc
    Exp e1, e2;
    public Plus(Exp e1, Exp e2)
      { this.e1=e1; this.e2=e2; }
    public override object Eval()
      { return (int)e1.Eval()+(int)e2.Eval(); }
  }
  public class Equals : Exp { // Likewise for Or, etc
    Exp e1, e2;
    public Equals(Exp e1, Exp e2)
      { this.e1=e1; this.e2=e2; }
    public override object Eval()
      { return (int)e1.Eval()==(int)e2.Eval(); }
  }
  public class Cond : Exp {
    Exp e1, e2, e3;
    public Cond(Exp e1, Exp e2, Exp e3)
      { this.e1=e1; this.e2=e2; this.e3=e3; }
    public override object Eval()
      { return ((bool)e1.Eval() ? e2 : e3).Eval(); }
  }
  public class Tuple : Exp {
    Exp e1, e2;
    public Tuple(Exp e1, Exp e2)
      { this.e1=e1; this.e2=e2; }
    public override object Eval()
      { return new Pair(e1.Eval(),e2.Eval()); }
  }
  public class Fst : Exp { // Likewise for Snd
    Exp e;
    public Fst(Exp e) { this.e=e; }
    public override object Eval()
      { return ((Pair)e.Eval()).fst; }
} }
```

Figure 1: Untyped expressions with evaluator

```
namespace T { // Typed expressions
  public class Pair<A,B> {
    public A fst; public B snd;
    public Pair(A fst, B snd)
      { this.fst=fst; this.snd=snd; }
  }
  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> { //Likewise for Snd
    Exp<Pair<A,B>> e;
    public Fst(Exp<Pair<A,B>> e) { this.e=e; }
    public override A Eval()
      { return e.Eval().fst; }
} }
```

Figure 2: Typed expressions with evaluator

typed version, particularly in an implementation that performs code specialization to avoid the cost of boxing integers and booleans [11].

The central observation of this paper is that the coding pattern used for `Exp` above has a strong connection with *generalized algebraic data types* (also called guarded recursive datatype constructors [25], or first-class phantom types [3, 8]). GADTs generalize the existing *parameterized algebraic datatypes* (PADTs) found in functional languages with support for constructors whose result is an instantiation of the datatype at types other than its formal type parameters. This corresponds to the subclass refinement feature used above. Case analysis over GADTs propagates information about the datatype instantiation into the branches. This corresponds, in part, to the refinement of signatures in overriding methods used above.

However, all is not rosy. As we shall see, many sophisticated GADT programs are easy to express, but sometimes even the simplest functions on ordinary parametric datatypes require either redundant runtime-checked casts or awkward higher-order workarounds. To illustrate the problem, take the presumably simpler task of coding up linked lists as an abstract generic class `List<T>` with `Nil<A>` and `Cons<A>` subclasses. Assuming the obvious definition of `Append`, a direct implementation of appending a lists of lists, the virtual method `Flatten`, requires an ugly cast (Figure 3). The root of the problem is that a virtual method must assume that the type of its receiver (`this`) is a generic instance of its class: it cannot impose nor make use of any pre-conditions on the class instantiation. Luckily, for functions over parameterized ADTs, such casts can always be avoided: we can re-code `Flatten` as a static method that uses a cast-free implementation of the *Visitor* pattern [6] to traverse its list, this time at a more specific instantiation.

Unfortunately, this limitation of virtual methods, which makes programming with parameterized datatypes merely inconvenient, means that some natural and safe GADTs programs can only be expressed by inserting redundant runtime-

```
public abstract class List<T> {...
  public abstract List<T> Append(List<T> that);
  public abstract List<U> Flatten<U>();
}
public class Nil<A> : List<A> {...
  public override List<U> Flatten<U>()
    { return new Nil<U>(); }
}
public class Cons<A> : List<A> {...
  A head; List<A> tail;
  public override List<U> Flatten<U>()
  { Cons<List<U>> This = (Cons<List<U>>) (object) this;
    return This.head.Append(This.tail.Flatten<U>()); }
}
```

Figure 3: `Flatten` on Lists, using casts

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

Figure 4: Equality on values, using casts

checked casts. Returning to the typed expression example, consider implementing an equality method on fully-evaluated expressions (literals and tuples)[1]. We add a virtual method `Eq` to `Exp<T>`, taking a single argument `that` of type `Exp<T>` and by default returning false. As is usual with a binary method such as `Eq`, we can implement it 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 4). This is clumsy and non-modular [2]. But there is another, more fundamental problem. Consider equality for instances of `Tuple`: the specialized code for equality on pairs in `TupleEq` cannot declare and make use of the fact that both `that` (of declared type `Tuple<C,D>`) and `this` will both actually have type `Tuple<A,B>` except through the use of a runtime-checked cast. The crux of the problem is that although information about a type instantiation is propagated through subclass refinement, there is still no way to constrain the type of the receiver. Here, the only caller of method `TupleEq<C,D>` uses the particular method instantiation `C=A,D=B` on a receiver of type `Exp<T>=Exp<Pair<A,B>>`. But, the override for `TupleEq<C,D>` cannot assume this callsite invariant and must assert it using a cast.

Now suppose that C♯ were extended to support *equational type constraints* on methods, as statically checked preconditions. Then adding the constraint `where T=Pair<C,D>` to the signature of `TupleEq` would allow us to restrict its callers, and so avoid the cast (Figure 5).

[1]It's not sensible to define equality between unevaluated typed `Exp<T>` (consider the case for `Fst`).

```
public abstract class Exp<T> { ...
  public virtual bool TupleEq<C,D>(Tuple<C,D> that)
    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> that)
    { return that.e1.Eq(e1) && that.e2.Eq(e2); }
}
```

Figure 5: Equality on values, using constraints

This approach also works for `Flatten`, allowing a direct, cast-free implementation (Figure 6), and demonstrating that our extension has more mundane applications than GADTs alone. For GADTs the situation is actually more dire than for PADTs: one cannot, in current C♯ or Java, define a type safe visitor pattern for certain GADTs, so the higher-order workaround no longer applies. In the absence of our extension, the GADT programmer *must* use casts.

```
public abstract class List<T> {...
 public abstract List<T> Append(List<T> that);
 public abstract List<U> Flatten<U>() where T=List<U>;
}
public class Nil<A> : List<A> {...
 public override List<U> Flatten<U>()
   { return new Nil<U>(); }
}
public class Cons<A> : List<A> {...
 A head; List<A> tail;
 public override List<U> Flatten<U>()// where A=List<U>
   { return this.head.Append(this.tail.Flatten<U>()); }
}
```

Figure 6: `Flatten`, using constraints

The contribution of this paper is threefold:

- We present a series of examples in C♯, demonstrating the utility of the GADT pattern for object-oriented programming languages that support generics, such as C♯, Java, C++, and Eiffel. We make the important observation that whilst all GADT *definitions* can be expressed, there are *programs* manipulating GADT values that cannot be written in current versions of C♯ and Java without the use of run-time casts.
- We identify a surprising expressivity gap compared to functional programming with parameterized algebraic datatypes (PADTs): operations with natural definitions in ML and Haskell require unnatural and non-extensible object-oriented encodings to ensure safety. With the introduction of generics, virtual dispatch is no longer as expressive as functional case analysis.
- To remedy these expressivity problems, we propose a generalization of C♯'s type parameter constraint mechanism. We also describe a generalization of `switch` to provide similar functionality to `case` for datatypes found in functional languages. Although a type based `switch` construct is present in Pizza and Scala[14, 13],

our typing rule is more expressive and accommodates GADTs. Both constructs are formalized as extensions to C♯ minor, a tiny subset subset of C♯ similar in style to FGJ [9]. We also prove a type soundness result.

The structure of the paper is as follows. Section 2 introduces the notion of GADT as proposed for functional languages. Section 3 describes informally the connection with object-oriented programming as exemplified by `Exp`, presents the two proposed extensions and discusses a Visitor pattern for GADTs. Section 4 presents a series of further examples in C♯. Section 5 presents the formalization. Section 6 discusses related work and Section 7 concludes.

2. GADTS IN FUNCTIONAL LANGUAGES

2.1 Datatypes

Functional programming languages such as Haskell and ML support 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 Haskell code that defines a binary tree parameterized on the type d of data and type k of keys stored in the nodes:

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

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

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

Notice how both term constructors have the fully generic result type *Tree k d*; there is no specialization of the type parameters to *Tree*.[2] 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 also have type *Tree k d*; this characteristic makes it a *regular* datatype.

Here is a lookup function for trees keyed on integers, defined by case analysis on a value of type *Tree Int d*, using Haskell's *pattern-matching* feature to switch on the datatype constructor and at the same time bind constructor arguments to variables:

$$\begin{array}{l}\mathit{find}\ ::\ \mathit{Int}\ \rightarrow\ \mathit{Tree}\ \mathit{Int}\ d\ \rightarrow\ \mathit{Maybe}\ d\\ \mathit{find}\ i\ t\ =\ \textbf{case}\ t\ \textbf{of}\\ \quad \mathit{Leaf}\ \rightarrow\ \mathit{Nothing}\\ \quad \mathit{Node}\ \mathit{key}\ \mathit{item}\ \mathit{left}\ \mathit{right}\ \rightarrow\\ \quad\quad \textbf{if}\ i = \mathit{key}\ \textbf{then}\ \mathit{Just}\ \mathit{item}\ \textbf{else}\\ \quad\quad\quad \textbf{if}\ i < \mathit{key}\ \textbf{then}\ \mathit{find}\ i\ \mathit{left}\ \textbf{else}\ \mathit{find}\ i\ \mathit{right}\end{array}$$

It is easy to type-check a function defined by case analysis such as this. Bound variables in patterns are assigned the formal types specified for constructor arguments in the datatype declaration, the type of each pattern is *unified* with the type of the scrutinee (here, revealing that formal type argument k of both *Node* and *Leaf* is *Int*), the branches are type-checked under this refined assumption, and then it is just necessary to check that each branch is assigned the same type, by unifying those types.

[2]The common, generic result types are forced by the Haskell syntax for PADT declarations — in GADT Haskell, every constructor declaration has its own explicit result type.

2.2 GADTs

Datatypes can be generalized in three ways:

1. The restriction that constructors all return 'generic' instances of the datatype can be removed. This is the defining feature of a GADT.

2. The regularity restriction can be removed, permitting datatypes to be used at different types within their own definition. In practice, to write useful functions over such types it is also necessary to support *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. The actual types at which such parameters are instantiated is not revealed by the type of the constructed term. This hiding of type arguments is, formally speaking, equivalent to adding *existential* quantification to the type system.

Most useful examples of GADTs make use of all three abilities. Consider the following implementation of the `Exp` type from Figure 2, written in a recent extension of Haskell with GADTs [17, 16]:

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

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 *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).

Now consider an evaluator for expressions, defined by case analysis on a value of type *Exp t*:

$$\begin{array}{l}\mathit{eval}\ ::\ \mathit{Exp}\ t\ \rightarrow\ t\\ \mathit{eval}\ e\ =\ \textbf{case}\ e\ \textbf{of}\\ \quad \mathit{Lit}\ i\ \rightarrow\ i\\ \quad \mathit{Plus}\ e1\ e2\ \rightarrow\ \mathit{eval}\ e1\ +\ \mathit{eval}\ e2\\ \quad \mathit{Equals}\ e1\ e2\ \rightarrow\ \mathit{eval}\ e1\ ==\ \mathit{eval}\ e2\\ \quad \mathit{Cond}\ e1\ e2\ e3\ \rightarrow\\ \quad\quad \textbf{if}\ \mathit{eval}\ e1\ \textbf{then}\ \mathit{eval}\ e2\ \textbf{else}\ \mathit{eval}\ e3\\ \quad \mathit{Tuple}\ e1\ e2\ \rightarrow\ (\mathit{eval}\ e1,\ \mathit{eval}\ e2)\\ \quad \mathit{Fst}\ e\ \rightarrow\ \mathit{fst}\ (\mathit{eval}\ e)\\ \quad \ldots\end{array}$$

Type checking of the **case** construct is not simply a matter of unifying the types of the branches, as was done for ordinary datatypes. (Indeed, combining *checking* of **case** on GADTs with full Haskell or ML type *inference* is much harder still, and is the subject of active research [17, 23].) The types of the branches in the **case** expression differ: for *Lit*, the type is *Int*, for *Equals*, it is *Bool*, whilst for *Tuple*, it is (a, b) for some type variables a and b. Fortunately, these types can be related to the declared type of the result (here:

t) by type-checking the branches under equational assumptions, namely to equate the type of the scrutinee (*Exp t*) to the result type of the constructors (*Exp Int*, *Exp Bool*, etc). These yield the equations shown below in comments following each branch.

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

Now consider equality on values, written in GADT Haskell rather than C♯ (cf. Figure 5). As above, we annotate the branches with the equations that are assumed:

```
eq :: (Exp t, Exp t) → Bool
eq (this, that) =    — this :: Exp t, that :: Exp t
  case this of
    Lit i →    — i :: Int, t = Int
      case that of
        Lit j → i == j    — j :: Int, t = Int
        _ → False
    Tuple e1 e2 →
      — e1 :: Exp a, e2 :: Exp b, t = (a, b)
      case that of
        Tuple f1 f2 →
          — f1 :: Exp c, f2 :: Exp d, t = (c, d)
          eq (e1, f1) && eq (e2, f2)
        _ → False
```

To type-check the outer branch for *Tuple* we assume the type equation $t = (a, b)$ and type assignment $e1 :: Exp\ a$, $e2 :: Exp\ b$. In the inner branch we assume $t = (c, d)$ and $f1 :: Exp\ c$, $f2 :: Exp\ d$ (generating fresh names for the type parameters to the *Tuple* constructor). Combining the equations on t we obtain $(a, b) = (c, d)$. From this, we derive $a = c$ 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 us 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 easier: all equations were of the form $t = \tau$ and there was no need to decompose constructed types.

3. GADTS IN C♯

We have now presented the GADT for `Exp` and associated operations `Eval` and `Eq` in C♯ (Section 1) and in Haskell (Section 2). Its definition in Haskell made use of all three features listed in Section 2.2 that characterize GADTs. We now consider these features in the context of the C♯ implementation. Feature (1) was expressed by defining a subclass of a generic type that did not just propagate the type parameters through to the subclass. (For example, `Plus` is a non-generic class that extends a generic class `Exp` at the particular instantiation `int`.) Feature (2) corresponds to the existence of fields in the subclass whose types are arbitrary instantiations of the generic type of the superclass. (For example, `Tuple` has fields of type `Exp<A>` and `Exp<B>` but a superclass of type `Exp<Pair<A,B>>`.) Feature (3) corresponds to the declaration of type parameters on the subclass that are not referenced in the superclass. (For example, `Fst<A,B>` has superclass `Exp<A>` that does not mention, nor reveal, `B`).

Let us now turn to the evaluator code. The Haskell *eval* function used case analysis; in C♯ (Figure 2) we used virtual dispatch to select the implementation of `Eval` appropriate to the constructor. The branches of Haskell's **case** construct were checked under assumptions equating types; in C♯ the signature of `Eval` specified in the `Exp` class was refined by substituting the actual type arguments specified for the superclass in place of the formal type parameters declared for `Exp`. This amounts to the same thing, when type equations are in solved form, assigning a type variable on one side of an equation to a type (its instantiation) on the other. This is the case for *eval*. For example, the branch for *Tuple* was checked under the assumption $t = (a, b)$. In C♯ the signature for the method `Tuple<A,B>.Eval` is obtained by applying the substitution `T` ↦ `Pair<A,B>` to the signature specified in the superclass.

Now consider the equality function. Both inner and outer *Lit* branches of the Haskell *eq* function are type-checked under the assumption $t = Int$, but this equation is not needed to type-check the expression $i == j$. In C♯ (Figure 4), the signature of `Eq` is refined in the `Lit` class to take an argument of type `Exp<int>`; this corresponds to applying the equation `T=int` as a substitution on the signature from the superclass; but, as in Haskell, this information is not needed for type-checking the body. In contrast, type-checking the *Tuple* branch does use the equational assumptions, as we saw at the end of Section 2: namely the equations $t = (a, b)$ and $t = (c, d)$. The C♯ code for `Eq` refines the signature with `T=Pair<A,B>` but then dispatches to `TupleEq`, *discarding* this information, which must be recovered with a redundant cast.

3.1 Equational constraints for C♯

As we will demonstrate in Section 4, a surprising number of GADT-manipulating programs can be written in C♯ and Java simply using the existing mechanisms of generics, subclassing and virtual dispatch. We have just seen an example of a program that can only be written through painful use of casts; moreover, we shall see in Section 3.2 that it is not possible to code a fully-general Visitor pattern. In Section 3.3 we will see that some natural functions over ordinary parameterized datatypes require contorted, or unsafe, implementations.

To remedy matters, we propose a modest extension of the existing *type constraint* mechanism supported by Java and C♯. In C♯, a type argument to a generic type or method can be required to satisfy a set of constraints, namely that the type extends some class or implements some interfaces. These constraints are specified by a `where` clause attached to the type or method declaration. For example:

```
class HashTable<K,D>
  where K : IHashable<K>, IEquatable<K> { ... }
class Array {
  static void Sort<T>(T[] a)
    where T : IComparable<T> {...a[i].CompareTo(p)...}
}
```

The constraints are *upper bounds* with respect to subtyping: they state that a type argument must be a subtype of the specified types. The first `where` clause above states that `HashTable<`τ`,`σ`>` is a valid type only if τ supports both `IHashable<`τ`>` and `IEquatable<`τ`>` interfaces; moreover, all methods defined in `HashTable` can assume this property of the type parameter. Similarly, an invocation `Array.Sort<`τ`>` is valid only if τ supports the interface `IComparable<`τ`>`, and code for `Sort` can rely on this. The language Scala [13] also supports lower bounds: the requirement that a type argument be a *supertype* of the specified type.

Our proposal is to extend the constraint language with *equational* constraints between types: the requirement that two types be equal for a generic instantiation to be valid. Unlike subtype constraints, there is no requirement that one of the types be a parameter of the enclosing method: we want the ability to impose additional constraints on *class* type parameters when declaring a *method* in that class. We saw this in the improved `TupleEq` method of Figure 5:

```
public abstract class Exp<T> {
  public virtual bool TupleEq<C,D>(Tuple<C,D> that)
    where T=Pair<C,D> { return false; }
}
```

Here the class type parameter `T` has been equated to a type `Pair<C,D>` that involves the method type parameters `C` and `D`. We will see more examples in the sections which follow.

In terms of language syntax, we propose simply to extend the grammar for `where`:

type-parameter-constraints-clause :
  `where` *type-parameter* `:` *type-parameter-constraints*
  `where` *type* `=` *type*

The C♯ type-checking rules are then extended as follows:

- **Use.** To successfully type-check the invocation of a method that has equational constraints, one must verify that the formal equations are satisfied when its actual class and method type arguments are substituted in. For example, to type-check the invocation `e.TupleEq<int,bool>(e2)` for receiver `e` of static type `Exp<Pair<int,bool>>` we simply check that the equation `T` = `Pair<C,D>` holds under the instantiation `T` $\mapsto$ `Pair<int,bool>`, `C` $\mapsto$ `int`, `D` $\mapsto$ `bool`.

- **Definition.** To type-check a method body that has equational constraints, we wish to take account of the equations when resolving overloading, checking assignment compatibility, performing method lookup, and so on. Potentially this is a complicated process, but there is a simpler approach: simply *solve* the constraints upfront. We can do this because if there is any substitution of type parameters that validates the equations, then there is a substitution – the *most general unifier* – that captures all such substitutions. We apply this substitution to the signature of the method, to the type of `this`, and to types occurring in the method body, and then type-check the body under those refined assumptions. A similar approach is used by Peyton Jones *et al.* to eliminate equations from the type system for GADT Haskell [17].

- **Overriding.** Subtype constraints in C♯ are 'inherited' by overriding methods and do not need to be redeclared. We adopt the same rule for equational constraints, but there is a new issue that we must address. It is possible for equations that are satisfiable at their virtual declaration to be unsatisfiable in their inherited form at the override, i.e. no instantiation validates the equations, and so the override is effectively dead. For example, consider the `Lit` class in Figure 5. If it were to override the `TupleEq` method described above, then it would inherit the `T=Pair<C,D>` equational constraint, but with `T` instantiated at `int` as its superclass is `Exp<int>`. There are no type arguments for `C` and `D` which validate `int=Pair<C,D>`, and so the method body can never be entered.

  We adopt the following rules:

  – We *prohibit* virtuals or overrides in which declared or inherited equations are, or have become, unsatisfiable.

  – We *allow* the concrete override of an abstract method or concrete implementation of an interface method to be omitted, but only when its inherited constraints would be unsatisfiable. Thus an abstract method no longer has a definition in all non-abstract subclasses, just in those with potential callers.

Note that we are relaxing the C♯ rules that mandate implementations for all interface methods and abstract virtuals in (non-abstract) subclasses: if the `TupleEq<A,B>` method had been declared `abstract`, the existing declaration of `Lit`, that does not provide a concrete implementation of `TupleEq`, would nevertheless be legal: unsatisfiability of the equation ensures that the absent implementation will never be missed.

To illustrate the type-checking process, take the `TupleEq` method overridden in the `Tuple` class (Figure 5). It inherits the constraint `T=Pair<C,D>` from its superclass, which, after substituting for `T`, is `Pair<A,B>=Pair<C,D>`. The most general unifier of this equation is `A` $\mapsto$ `C`, `B` $\mapsto$ `D`, and applying this substitution assigns `this` the type `Tuple<C,D>`, allowing the body to be type-checked.

3.2 A Visitor pattern for GADTs

The Interpreter design pattern used to implement `Exp` has the disadvantage that code for a particular operation such as `Eval` is spread across the various node classes. A popular alternative is to package the operations together in a *visitor* object, and to define an *acceptor* method on the expression class that takes a visitor object as argument and then dispatches to the appropriate operation as determined by the node class [6]. Typically the visitor methods are packaged as an interface type. For the untyped expression language of Figure 1 this might be the following, shown here with an illustrative acceptor method:[3]

```
public interface IExpVisitor<R> {
  R VisitLit(Lit e);
  R VisitPlus(Plus e);
  R VisitEquals(Equals e);
  R VisitCond(Cond e);
  R VisitTuple(Tuple e);
  R VisitFst(Fst e);
}
```

[3]It's possible to utilize overloading and use the name `Visit` for all methods but this would obscure the explanation.

```
public interface IExpVisitor<T,R> {
  R VisitLit(Lit e) where T=int;
  R VisitPlus(Plus e) where T=int;
  R VisitEquals(Equals e) where T=bool;
  R VisitCond<A>(Cond<A> e) where T=A;
  R VisitTuple<A,B>(Tuple<A,B> e) where T=Pair<A,B>;
  R VisitFst<A,B>(Fst<A,B> e) where T=A;
}
public abstract class Exp<T> { ...
  public abstract R Accept<R>(IExpVisitor<T,R> v);
  public T Eval(){return Accept(new EvalVisitor<T>();)}
}
public class Lit : Exp<int> { ...
  public override R Accept<R>(IExpVisitor<int,R> v)
    { return v.VisitLit(this); }
}
public class Plus : Exp<int> { ...
  public override R Accept<R>(IExpVisitor<int,R> v) {
    { return v.VisitPlus(this); }
}
public class Equals : Exp<int> { ...similar to Plus...}
public class Cond<T> : Exp<T> { ...
  public override R Accept<R>(IExpVisitor<T,R> v) {
    { return v.VisitCond<T>(this); }
}
public class Tuple<A,B> : Exp<Pair<A,B>> { ...
  public override R Accept<R>
    (IExpVisitor<Pair<A,B>,R> v)
    { return v.VisitTuple(this); }
}
public class Fst<A,B> : Exp<A> { ...
  public override R Accept<R>(IExpVisitor<A,R> v)
    { return v.VisitFst(this); }
}
public class EvalVisitor<T> : IExpVisitor<T,T> {
  public T VisitLit(Lit e) { return e.value; }
  public T VisitPlus(Plus e)
    { return e.e1.Eval() + e.e2.Eval(); }
  public T VisitEquals(Equals e)
    { return e.e1.Eval() == e.e2.Eval(); }
  public T VisitCond<A>(Cond<A> e)
    { return e.e1.Eval() ? e.e2.Eval() : e.e3.Eval(); }
  public T VisitTuple<A,B>(Tuple<A,B> e)
    { return new Pair<A,B>(e.e1.Eval(), e.e2.Eval()); }
  public T VisitFst<A,B>(Fst<A,B> e)
    { return e.e.Eval().fst; }
}
```

Figure 7: Typed visitor interface for expressions with evaluator visitor

```
public abstract class Exp { ...
  public abstract R Accept<R>(IExpVisitor<R> v);
}
public class Fst : Exp { ...
  public override R Accept<R>(IExpVisitor<R> v)
    { return v.VisitFst(this); }
}
```

We have parameterized the visitor interface on the result type of the visitor methods: for example, a type-checker visitor might return a `bool`, whilst an evaluator visitor would return an `object`. Here, for example, is part of the code for an evaluator visitor:

```
public class EvalVisitor : IExpVisitor<object> {
  static EvalVisitor evalVis = new EvalVisitor();
  public static object Eval(Exp e)
    { return e.Accept(evalVis); }
  public object VisitFst(Fst fstexp)
    { return ((Pair)Eval(fstexp.e)).fst; } ...
}
```

To adapt this to the typed, GADT variant of expressions from Figure 2 we can abstract over the type parameters of the constructors, as follows:

```
public interface IExpVisitor<R> {
  R VisitLit(Lit e);
  R VisitPlus(Plus e);
  R VisitEquals(Equals e);
  R VisitCond<A>(Cond<A> e);
  R VisitTuple<A,B>(Tuple<A,B> e);
  R VisitFst<A,B>(Fst<A,B> e);
}
public abstract class Exp<T> { ...
  public abstract R Accept<R>(IExpVisitor<R> v);
}
public class Fst<A,B> : Exp<A> { ...
  public override R Accept<R>(IExpVisitor<R> v)
    { return v.VisitFst(this); }
}
```

Unfortunately, this interface is not sufficiently refined to implement statically-typed visitors: we are forced to use casts. Consider part of the evaluator visitor:

```
public class EvalVisitor<T> : IExpVisitor<T> {
  // We know that T=int but the compiler does not!
  public T VisitLit(Lit e)
    { return (T) (object) e.value; }
  ...
}
```

The problem is that we have not expressed (a) the fact that there is a relationship between the result type of the visitors and the type argument of `Exp` (namely, they're the same), and (b) that the type argument is determined by the particular subclass of `Exp` passed to the visitor methods. Figure 7 presents the solution: parameterize the visitor interface on the return type `R` *and* the expression type `T`, and express the fact that `T` is related to the type parameters of the node types through equational constraints. No casts required!

Observant readers may have noticed that there is some redundancy in the interface: method type parameters that are identified with the type parameter of `Exp` can be removed:

```
public interface IExpVisitor<T,R> {
  ...
  R VisitCond(Cond<T> e);
  R VisitFst<B>(Fst<T,B> e);
}
```

We can obtain a visitor interface from any GADT using the following recipe:

- For a class `C<`$X_1, \ldots, X_n$`>` declare a visitor interface

  `interface IVisC<`$X_1, \ldots, X_n$`,R>`

  and add an abstract `Accept` method to `C`:

  `abstract R Accept<R>(IVisC<`$X_1, \ldots, X_n$`,R> v);`

- For each class `D<`$Y_1, \ldots, Y_m$`>` extending `C<`$T_1, \ldots, T_n$`>`, declare a visitor method on the interface

  `R VisitD<`$Y_1, \ldots, Y_m$`>(D<`$Y_1, \ldots, Y_m$`> arg)`
  `where` X_1`=`$T_1, \ldots, X_n$`=`T_n`;`

  and override the `Accept` method in `D`:

```
class D<Y1,...,Ym> :  C<T1,...,Tn> {
  override R Accept<R>(IVisC<T1,...,Tn,R> v) {
    return v.VisitD(this);
  }
}
```

```
public interface IListVis<T, R> {
  R VisitNil(Nil<T> n);
  R VisitCons(Cons<T> c);
}
public abstract class List<T> {
  public abstract List<T> Append(List<T> l);
  public abstract R Accept<R>(IListVis<T,R> v);
  public static List<U> Flatten<U>(List<List<U>> l)
    { return l.Accept(new FlattenVis<U>()); }
}
public class Nil<A> : List<A> {
  public override List<A> Append(List<A> l)
    { return l; }
  public override R Accept<R>(IListVis<A,R> v)
    { return v.VisitNil(this); }
}
public class Cons<A> : List<A> {
  public A head; public List<A> tail;
  public Cons(A head, List<A> tail) {
    this.head=head; this.tail=tail;
  }
  public override List<A> Append(List<A> l) {
    return new Cons<A>(this.head, this.tail.Append(l));
  }
  public override R Accept<R>(IListVis<A,R> v)
    { return v.VisitCons(this); }
}
public class FlattenVis<U> : IListVis<List<U>,List<U>>{
  public List<U> VisitNil(Nil<List<U>> n)
    { return new Nil<U>(); }
  public List<U> VisitCons(Cons<List<U>> c)
    { return c.head.Append(c.tail.Accept(this)); }
}
```

Figure 8: Generic Lists

- Optionally, equations of the form $\texttt{X}_i\texttt{=Y}_j$ can be omitted, along with type parameter $\texttt{Y}_j$; uses of $\texttt{Y}_j$ in the method signature must then be replaced with $\texttt{X}_i$.

Observe that, for a parameterized datatype (PADT), this optimization yields a visitor interface that makes no use of constraints or of type parameters to methods.

3.3 Revisiting datatypes in C♯

Given that we can express many, though not all, GADT programs, can we justify extending C♯ just to capture a few more exotic examples? It turns out that our inability to express certain GADT programs is a symptom of a more fundamental problem with the design of virtual methods in generic classes, present in both C♯ and Java. To illustrate the deficiency, we revisit the implementation of the list library sketched in Section 1.

Figure 8 shows a simple implementation of generic lists, with a single abstract class `List<T>` and two concrete subclasses `Nil<A>` and `Cons<A>`. Observe that `List<T>` is not a GADT *per se*, but an ordinary parameterized algebraic datatype (PADT) (the subclasses are as generic as the superclass and do not specialize it). This time, we present the full definition of `List<T> Append(List<T> l)` that uses virtual dispatch to do case analysis on the receiver, and a cast-free implementation of `Flatten`. The safe `Flatten` method is extremely clumsy. To avoid casting, the programmer is forced to introduce an (optimized) variant of the Visitor pattern described in Section 3.2. Not only is this very verbose, but it also has another drawback: the behavior of `Flatten` cannot be extended to future subclasses without modifying the code for the `IVisitor` interface. Contrast this with `Append`, whose implementation is extensible by using method override in any future subclass of `List<T>`.

Of course, with dynamic casting we can give the more direct implementation of `Flatten` that does use a virtual method (Figure 3). This at least has the merit of being extensible, but is also unnecessarily expensive. Worse though, it is potentially unsafe in the sense that there is no way to prevent the cast from failing on some receivers: calling `l.Flatten<int>()` on an object of the wrong static type, say `List<int> l;`, is legal, but will result in a runtime exception on entry to the method. In the equivalent Java program, the cast cannot be checked at runtime due to the erasure semantics of Java generics (a Java compiler will issues a warning): calling `l.Flatten<string>()` on a receiver of the wrong static type, say `List<List<Int>> l;`, is legal, but will not raise an exception until the elements of the flattened list are accessed as strings (since they are integers).

Why is a safe implementation using virtual methods impossible? The root of the problem is that an override of a virtual method can only assume that the type of its receiver (`this`) is a generic instance of its class, it may not make any additional assumptions about the instantiation of its class. That is the crucial difference between `Append` and `Flatten`: the overrides of `Append` happen to be completely generic in the receiver's formal type argument (`A`), so their bodies type check. A virtual method implementation of `Flatten`, on the other hand, would need to know that the instantiation of its class is itself a list of elements, so that its head can be appended to its flattened tail. The virtual methods of C♯ and Java are not flexible enough to express such refinements of the class instantiation, requiring tedious workarounds. Of course, the problem isn't peculiar to `Flatten`: non-generic functions like `Sum`, that adds the elements in a list of *integers*, generic functions like `Unzip`, that splits a list of *pairs*, and indeed most instantiation specific functions on PADTs are equally awkward to code.

Equational constraints allow us to write the cast-free, safe and extensible implementation of `Flatten` in Figure 6. The abstract `Flatten` method is qualified by an equational constraint relating the class type parameter `T` to the method type parameter `U`. This precisely states the requirement that any `Flatten<U>`-receiver of type `List<T>`, must satisfy `T = List<U>`, constraining the type of the receiver to a subtype of `List<List<U>>`. Enforcing this restriction at method call sites allows the type system to assume it holds within the overrides of `Flatten<U>`. With a bit of equational reasoning, the type system can check the cast-free override of `Flatten<U>` in `Cons<A>` is safe: assuming the type equality `A = List<U>`, appending to the head of the cons cell is safe, because the head not only has type `A`, but also `List<U>`; flattening the tail of the cons cell is safe since the tail not only has type `List<A>`, but also `List<List<U>>`.

3.4 Generalizing switch

Sometimes its just more convenient and clear to sacrifice the extensibility provided by virtual methods and directly dispatch on the type of an object using an inline `switch` construct, analogous to Haskell's **case** construct. The main advantage of an inline dispatch is that it gathers all the possible continuations of the test into a single, shared scope. Having direct access to any outer variables relieves the programmer from the tedious (and error-prone) chore of closing every continuation over its free variables, as required

when abstracting dispatch into separate overrides of a virtual method or an implementation of the Visitor Pattern.

Although Pizza [14] (and now Scala [13]) contained a similar construct, its typing rules were not formalized. Our contribution is to present new typing rules for `switch` that are more expressive, deriving and exploiting equational constraints between types particular to each branch.

For example, the `Eval` code of Figure 2 and `Eq` code of Figure 5 can be re-written as static methods that switch on their arguments:

```
public static T Eval<T>(Exp<T> exp) {
  switch (exp) {
    case Lit e :
      return e.value;
    case Plus e :
      return Eval(e.e1) + Eval(e.e2);
    case Equals e :
      return Eval(e.e1) == Eval(e.e2);
    case Cond<A> e :
      return Eval(e.e1) ? Eval(e.e2) : Eval(e.e3);
    case Tuple<A,B> e :
      return new Pair<A,B>(Eval(e.e1), Eval(e.e2));
    case Fst<A,B> e :
      return Eval(e.e).fst;
  }
}
public static bool Eq<T>(Exp<T> e1, Exp<T> e2) {
  switch (e1, e2) {
    case (Lit x, Lit y) :
      return x.value == y.value;
    case (Tuple<A,B> x, Tuple<C,D> y) :
      return Eq(x.fst, y.fst) && Eq(x.snd, y.snd);
    default :
      return false;
  }
}
```

In detail, our extension splits into two pieces: (1) support for switching on multiple expressions, used in `Eq` above; and (2) the ability to match against a class, bind its type parameters, and type-check the branch under equational assumptions about of the type of the switch expression.

This is much more concise than spreading the code across the classes, (`Eq<T>` would otherwise require 3 virtual methods with 4 overrides), though it shares with the Visitor pattern [6] the lack of extensibility and loss of encapsulation (we must either weaken access qualifiers on the fields of the subclasses, or provide accessor methods).

The obvious Pizza (and Scala) translation of `Eval<T>` type checks, but the translation of `Eq<T>` does not: checking the branch comparing an instance of `Tuple<A,B>` with an instance of `Tuple<C,D>` relies on our switch construct's novel use of type equations.

Syntactically, our `switch` construct extends the existing C♯ grammar, as shown in Figure 9 (changes in grey).

A single `case` *match* has the typical form $\texttt{C<X}_1, \ldots, \texttt{X}_n\texttt{>}\ x$. It both binds formal type parameters $\texttt{X}_1, \ldots, \texttt{X}_n$, and declares a formal argument x of static type $\texttt{C<X}_1, \ldots, \texttt{X}_n\texttt{>}$: both are scoped locally to the statement list of the enclosing "switch-section" or branch. We call $\texttt{C<X}_1, \ldots, \texttt{X}_n\texttt{>}$ the *type pattern* of the match.

To type-check a switch statement, we first determine the static type of each expression in the expression list. Let $\texttt{D<T}_1, \ldots, \texttt{T}_k\texttt{>}$ refer to the static type of the ith expression in the expression list. The statement list of each branch is checked in a scope determined as follows. For the ith pattern in the branch, we check that the formal type pattern, $\texttt{C<X}_1, \ldots, \texttt{X}_n\texttt{>}$, is derived from some (open) instantiation of $\texttt{D<U}_1, \ldots, \texttt{U}_k\texttt{>}$ (by chasing the inheritance hierarchy upwards from $\texttt{C<X}_1, \ldots, \texttt{X}_n\texttt{>}$ to some formal instantiation of `D`). If it is not (because `C` does not derive from `D`), the match is ill-formed and produces a compile-time error. Otherwise, we add the equation $\texttt{D<T}_1, \ldots, \texttt{T}_k\texttt{>} = \texttt{D<U}_1, \ldots, \texttt{U}_k\texttt{>}$ to the set of equations for this branch. We add one equation per corresponding expression and match in the expression and match lists (which must have the same length). Once we have gathered all the equations for the branch, we unify them to see if they have any solution. If not, the branch is dead, and the compiler could either issue an error (ruling out dead code), or a warning (allowing dead code) and skip type-checking the unreachable branch. If the equations have a solution, they must also have a most-general one. We type-check the statement-list of the branch assuming the type equalities induced by that most-general solution as well as the local declaration of the variable x from each match. In practice, a compiler could either substitute the solutions through the scope of the branch and its statement list, or cache the substitution, applying it whenever required to test type compatibility. Note that each branch must be checked independently of the equations induced by other branches, and that the outer scope as well as the return type of the enclosing method may itself be specialized by the unifier of each branch. In particular, this allows different branches to `return` from the method with values of different types. The default branch is checked in the scope of the entire switch statement, with no additional refinement of the scope. Because matches are binding constructs that extend the outer scope, it is illegal to jump directly into a branch, by-passing the switch.

A switch statement with n expressions in its expression list is executed as follows. The expressions are evaluated to

switch-statement :
  `switch (`*expression*`)` *switch-block*
  `switch (`*expression-list*`)` *switch-block*
switch-block :
  `{` *switch-sections*$_{opt}$ `}`
switch-sections :
  switch-section
  switch-sections switch-section
switch-section :
  switch-labels statement-list
  `case` *match-expression* `:` *statement-list*
switch-labels :
  switch-label
  switch-labels switch-label
switch-label :
  `case` *constant-expression* `:`
  `default :`
match-expression :
  match
  `(` *match-list* `)`
match :
  identifier type-parameter-list$_{opt}$ *identifier*$_{opt}$
match-list :
  match
  match `,` *match-list*

Figure 9: Extensions to `switch`

objects $o_1, ..., o_n$. If any of the values is null, we immediately enter the default branch. Otherwise, the branches are tested sequentially. The current branch is taken if, and only if, for each $i \leq n$, the runtime type of object o_i is compatible with the type pattern, `C<`$\mathtt{X}_1, \ldots, \mathtt{X}_n$`>`, of the ith match of the branch, for some actual instantiation $\mathtt{T}_1, \ldots, \mathtt{T}_n$, of the formal type parameters in $\mathtt{X}_1, \ldots, \mathtt{X}_n$. If all of the matches are compatible, the actual instantiation of each type pattern is bound to that pattern's formal type parameters, the object o_i is bound to the variable of the ith match, and the `case` block is entered. Otherwise, we proceed to the next branch, falling through to the default case when no branch is taken. For type safety, it is vital that a non-default branch is only entered if all of the objects are non-null: it is precisely the dynamic test against a non-null value that justifies the type equations used to check each branch.

Interestingly, it is easier to compile our `switch` in the type-erasing interpretation of generics found in Java: to compile a match against `C<`$\mathtt{X}_1, \ldots, \mathtt{X}_n$`>` x one would simply generate a test `x instanceof C`, as type arguments are erased at runtime. For C♯, it is necessary to use reflective capabilities to (a) test the class of the object, independent of its generic instantiation, and (b) bind the instantiation to type parameters, probably by invoking a generic method whose body contains code for the branch. Ideally, the Common Language Runtime could be extended to support this match-and-bind primitive directly. Adapting the techniques of [11] would require code specialization at the level of branches, not just methods, since existential type parameters may only be discovered on branching.

4. EXAMPLES

In this section we present a number of examples, already described in the literature on GADTs, but now presented as programs in C♯.

4.1 Statically typed printf

The libraries supplied with C♯ and the most recent release of Java provide methods similar to the `printf` function well-known to C programmers, used for formatted output of a list of arguments. For example, here is its simplest variant in C♯ from the `System.String` class:

```
string Format(string format, params object[] args);
```

This approach to formatting is preferable to *ad hoc* appending of strings, because style (the format string `format`) is separated from content (the arguments `args`). The drawback is that static type safety is lost: it is not possible to check statically that the number and types of placeholders in `format` match the number and types of `args`.

But suppose we use a GADT for the format specifier in place of a string [25, 8]. Figure 10 presents code in C♯.

The `Format<A>` generic class represents formatters that produce a value of type `A`. Formatters for integers (`Int`), characters (`Char`), and string literals (`Lit`) are presented. Constructors for each of these take another formatter as argument, representing the remainder of the format, and in the case of literals, the literal in question. Formatters are chained together, ending with a use of the `Stop` formatter. For conciseness, some trivial helper functions `I`, `C` and `S` are defined; type inference for generic methods then saves an abundance of angle brackets. The expression

```
S("int i = ", I(S(" and char c = ", C(stop))))
```

```
delegate B Function<A,B>(A arg);
public abstract class Format<A> {
  public abstract A Do(StringBuilder b);
}
public class Int<A> : Format<Function<int,A>> {
  Format<A> f; public Int(Format<A> rest) { this.f=f; }
  public override Function<int,A> Do(StringBuilder b)
    { return delegate(int i)
        { return f.Do(b.Append(i)); };
    }
}
public class Char<A> : Format<Function<char,A>> {
  Format<A> f; public Char(Format<A> f) { this.f=f; }
  public override Function<char,A> Do(StringBuilder b)
    { return delegate(char c)
        { return f.Do(b.Append(c)); };
    }
}
public class Lit<A> : Format<A> {
  string s; Format<A> f;
  public Lit(string s, Format<A> f)
    { this.s=s; this.f=f; }
  public override A Do(StringBuilder b)
    { return f.Do(b.Append(s)); }
}
public class Stop : Format<string> {
  public override string Do(StringBuilder b)
    { return b.ToString(); }
}
public class Helper {
  static Int<A> I<A>(Format<A> f)
    { return new Int<A>(f); }
  static Char<A> C<A>(Format<A> f)
    { return new Char<A>(f); }
  static Lit<A> S<A>(string s, Format<A> f)
    { return new Lit<A>(s,f); }
  static Stop stop = new Stop();
  public static void Main() {
    Format<Function<int,Function<char,string>>> fmt =
      S("int i = ", I(S(" and char c = ", C(stop))));
    string out = fmt.Do(new StringBuilder())(34)('a');
    Console.WriteLine(out);
  }
}
```

Figure 10: Printf

is the equivalent of the printf-style format string

```
"int i = %d and char c = %c".
```

The clever bit is its type:

```
Format<Function<int,Function<char,string>>>,
```

which describes a formatter that yields a function that accepts an integer, then a character, and returns a string.

4.2 Types as values

One motivation for some of the previous work on GADTs was to obtain runtime representations of types as values [25]. These can then be used to mimic dynamic typing in fully-statically-typed languages such as Haskell, without destroying existing properties of the language. They also pave the way for writing so-called "polytypic" functions that analyze the structure of types at runtime.

Many object-oriented languages support runtime types already. Despite this, it is instructive to study their encoding as GADTs. The existing runtime type capability has some drawbacks: it is intrusive (*all* objects carry runtime type information), and incomplete (in Java, generic type arguments

```
public abstract class Rep<T> {
  public abstract bool Eq(T x, T y);
  public abstract string Pretty(T x);
}
public class IntRep : Rep<int> {
  public override bool Eq(int x, int y)
    { return x==y; }
  public override string Pretty(int x)
    { return x.ToString(); }
}
public class BoolRep : Rep<bool> {
  public override bool Eq(bool x, bool y)
    { return x==y; }
  public override string Pretty(bool x)
    { return x ? "true" : "false"; }
}
public class PairRep<A,B> : Rep<Pair<A,B>> {
  Rep<A> a; Rep<B> b;
  public PairRep(Rep<A> a, Rep<B> b)
    { this.a = a; this.b = b; }
  public override bool Eq(Pair<A,B> x, Pair<A,B> y)
    { return a.Eq(x.fst, y.fst) && b.Eq(x.snd,y.snd); }
  public override string Pretty(Pair<A,B> p)
    { return "(" + a.Pretty(p.fst)
           + "," + b.Pretty(p.snd) + ")"; }
}
public class IEnumRep<T> : Rep<IEnumerable<T>> {
  Rep<T> rep;
  public IEnumRep(Rep<T> rep) { this.rep=rep; }
  public override bool Eq(IEnumerable<T> x,
                          IEnumerable<T> y) {
    IEnumerator<T> xenum = x.GetEnumerator();
    IEnumerator<T> yenum = y.GetEnumerator();
    while (xenum.MoveNext() && yenum.MoveNext())
      if (!rep.Eq(xenum.Current, yenum.Current))
        return false;
    return !xenum.MoveNext() && !yenum.MoveNext();
  }
  public override string Pretty(IEnumerable<T> x) {
    string sep = ""; string result = "";
    foreach (T xitem in x)
      { result += sep + rep.Pretty(xitem); sep = ","; }
    return result;
  }
}
```

Figure 11: Types as values

are lost through type erasure; in C♯ they are preserved, but it is not possible to deconstruct constructed types, binding type arguments to type variables at runtime).

The idea behind types-as-values is to represent a type τ as a *value* of type `Rep<`τ`>`. Figure 11 presents a generic class `Rep<T>` whose subclasses represent a number of C♯ types: `int`, `bool`, pairs and instantiations of `IEnumerator`. Also illustrated are two polytypic functions: equality, and a pretty-printer function. These functions dispatch on the type representation to code specialized for that type. Note that this is *not* possible using the existing runtime type features of C♯ or Java: `Pretty` on `Rep<IEnumerator<`τ`>>` looks at the representation of τ to determine how to pretty-print enumerated items, and `Eq` expresses statically the fact that its arguments have the same type, neither of which can be captured by runtime types in Java and C♯.

There are three interesting facets to `Rep`:

- It is an example of a *phantom* type: its type parameter is not used to type data, but is used to force type-distinctions.

```
// Represent natural numbers using classes
public class Nat { }
public class Zero : Nat { }
public class Succ<T> : Nat { }
public delegate B Function<A,B>(A arg);
// Lists of A with length L
public abstract class List<A,L> {
  public abstract A Head<K>() where L=Succ<K>;
  public abstract List<A,K> Tail<K>() where L=Succ<K>;
  public abstract List<B,L> Map<B>(Function<A,B> f);
  public abstract List<Pair<A,B>,L> Zip<B>(List<B,L> l);
}
public class Nil<A> : List<A,Zero> {
  public override List<B,Zero>
    Map<B>(Function<A,B> f) { return new Nil<B>(); }
  public override List<Pair<A,B>,Zero>
    Zip<B>(List<B,Zero> that)
    { return new Nil<Pair<A,B>>(); }
}
public class Cons<A,L> : List<A,Succ<L>> {
  public A head; public List<A,L> tail;
  public Cons(A head, List<A,L> tail)
    { this.head=head; this.tail=tail; }
  public override A Head<K>() { return head; }
  public override List<A,K> Tail<K>() { return tail; }
  public override List<B,Succ<L>>
    Map<B>(Function<A,B> f)
    { return new Cons<B,L>(f(head), tail.Map(f)); }
  public override List<Pair<A,B>,Succ<L>>
    Zip<B>(List<B,Succ<L>> that)
    { return new Cons<Pair<A,B>,L>(
        new Pair<A,B>(head, that.Head<L>()),
        Zip(that.Tail<L>())); }
}
```

Figure 12: Sized lists, using equational constraints

- It is an example of a *type-indexed* datatype, and `Eq` and `Pretty` are examples of type-indexed functions. Values of type `Rep<`τ`>` are determined by the structure of τ, and the behavior of `Eq` and `Pretty` is likewise determined by the structure of τ.

- Moreover, it is a *singleton* type: for each τ, there is (at most) one value of type `Rep<`τ`>`, if we neglect object identity and the existence of `null`.

In Section 4.5 we present an extended example that makes use of `Rep` to type-check the little language of Section 1.

4.3 Sized lists

Our next example (Figure 12) uses a 'phantom' type parameter `L` to a list `List` datatype to encode the length of the list in the type. Observe in particular how the equational constraints on the `Head` and `Tail` methods force the list to be non-empty; the `Nil` class need not override these methods because the constraint can never be satisfied in that subclass. The method signatures of `Map` and `Zip` express an invariant: the input and output lists have the same length.

It is even possible to assign a type to a size-correct `Append` operation. However, the operation would need take a third argument, an instance of an auxiliary `Sum` class that 'witnesses' the fact that the length of the resulting list is the sum of the lengths of the arguments lists and is used to drive the computation.

More sophisticated invariants on various data structures can be encoded using GADTs, for example, invariants associated with binomial heaps and red-black trees [22, 21].

However, it is probably too early to say whether these encodings are truly practical for large-scale programming.

4.4 Typed expressions with environments

Suppose that we wish to add variables and a local binding construct to the little language of Section 1:

$$exp ::= \quad \ldots \mid var \mid \texttt{let}\ var = exp\ \texttt{in}\ exp$$

To implement an interpreter we now need to evaluate expressions in the context of an environment that maps variables to values. At first it would appear that we must abandon static typing, as values in the environment will be of different types. If we had only the types `int` and `bool` then we could simply split the environment into two, but in general we cannot do this. So instead, we parameterize expressions by both the type denoted by the expression, *and* by the type of environment in which it must be evaluated [22]. Figure 13 presents code that does just that, representing variables using a GADT encoding of the natural numbers in order to index the environment.

4.5 Type checking

Our final example in Figure 14 brings everything together. The `TC` virtual method *type-checks* an untyped `Exp` expression (Figure 1) to produce a typed expression `Exp<T>` (Figure 2) paired with a type representation `Rep<T>` (Figure 11) encapsulated in a class `AnyTypedExp` that hides the type parameter `T`. `TC` just returns null if there is a type error. Type checking proceeds by dispatch on the untyped expression but uses our extended `switch` construct to inspect the result of recursing on subexpressions and to inspect type representations. Note the cunning use of the `Equal` GADT, used to 'witness' equality of type representations. This example is based on a Haskell implementation by Weirich [24].

5. FORMALIZATION

The aim of this section is provide evidence that our informally described extensions of C♯ are sound. We formalize the extensions for a small, but representative, fragment of C♯, and prove a type soundness theorem using standard operational techniques. After presenting the type system and operational semantics, we prove the usual *Preservation* and *Progress* theorems (Theorems 2 and 3) that establish *Type Soundness* (Theorem 4). Preservation tells us that program evaluation preserves types. Progress tells us that well-typed programs are either already fully evaluated, may be evaluated further, or are stuck, but only at the evaluation of an illegal cast (but not, say, at an undefined runtime member lookup). The fact that we have to accommodate stuck programs has nothing to do with our extensions; it is just the usual symptom of supporting runtime-checked down casts.

We formalize our proposed extensions in 'C♯ minor' [10], a small, purely-functional subset of C♯ version 2.0 [20, 7]. Its syntax, typing rules and small-step reduction semantics are presented in Figures 15 and 16. To aid the reader, we emphasize the essential differences to (constraint-free) C♯ minor using shading. This formalization is based on Featherweight GJ [9] and has similar aims: it is just enough for our purposes but does not "cheat" – valid (equation-free) programs in C♯ minor really are valid C♯ programs. The differences from Featherweight GJ are as follows:

- There are minor syntactic differences between Java and C♯: 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`. (In the full language, redeclaration of an inherited method as `virtual` introduces a new method without overriding the inherited one. Our subset does not support this.)
- For simplicity, we omit bounds on type parameters. Instead, we extend the language with *equations* on types, specified at virtual method definitions and implicitly inherited at method overrides.
- We include a separate rule for subsumption instead of including subtyping judgments in multiple rules.
- We fix the reduction 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, for added drama, runtime casts.

For readers unfamiliar with the work on Featherweight GJ we summaries the language here; for more details see [9].

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 and type signature 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 $\overline{T}$, a (possibly empty) sequence of type equations $\overline{E}$, and a body consisting of a single statement, `return` e`;`. The equation-less sugar

$$Q\ T\ m\texttt{<}\overline{X}\texttt{>(}\overline{T}\ \overline{x}\texttt{)}\ \{\texttt{return}\ e\texttt{;}\}$$

abbreviates a declaration with an empty `where` clause ($|\overline{E}| = 0$). By design, the typing rules only allow equations to be placed on a *virtual* method definition: equations are inherited, modulo base-class instantiation, by any overrides of this virtual method. Implicitly inheriting equations matches C♯'s implicit inheritance of bounds on type parameters.[4]

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 `new` $I\texttt{(}\overline{e}\texttt{)}$. A **value** v is a fully-evaluated expression and (always) has the form `new` $I\texttt{(}\overline{v}\texttt{)}$.

A **class table** $\mathcal{D}$ maps class names to class definitions. The distinguished class `object` is not listed in the table and is dealt with specially.

[4] An alternative would be to require the explicit redeclaration of any inherited constraints.

```
// Environments are either empty,
// or pair a value of type T with the rest of the environment, of type E
public class EnvNil { }
public class EnvCons<T,E> {
  public T t; public E e; public EnvCons(T t, E e) { this.t=t; this.e=e; }
}
// Expressions have type T in context of an environment of type E
public abstract class Exp<E,T> {
  public abstract T Eval(E env);
}
public abstract class Var<E,T> : Exp<E,T> { }
public class VarZero<E,T> : Var<EnvCons<T,E>,T> {
  public override T Eval(EnvCons<T,E> env) { return env.t; }
}
public class VarSucc<E,T,T2> : Var<EnvCons<T2,E>,T> {
  Var<E,T> v; public VarSucc(Var<E,T> v) { this.v = v; }
  public override T Eval(EnvCons<T2,E> env) { return v.Eval(env.e); }
}
public class Lit<E> : Exp<E,int> {
  int value; public Lit(int value) { this.value=value; }
  public override int Eval(E env) { return value; }
}
// Plus, Or etc similar
public class Cond<E,T> : Exp<E,T> {
  Exp<E,bool> e1; Exp<E,T> e2, e3;
  public Cond(Exp<E,bool> e1, Exp<E,T> e2, Exp<E,T> e3) { this.e1=e1; this.e2=e2; this.e3=e3; }
  public override T Eval(E env) { return e1.Eval(env) ? e2.Eval(env) : e3.Eval(env); }
}
public class Let<E,A,B> : Exp<E,B> {
  Exp<E,A> e1; Exp<EnvCons<A,E>,B> e2;
  public Let(Exp<E,A> e1, Exp<EnvCons<A,E>,B> e2) { this.e1=e1; this.e2=e2; }
  public override B Eval(E env) { return e2.Eval(new EnvCons<A,E>(e1.Eval(env), env)); }
}
```

Figure 13: Typed expressions in typed environments

A typing environment Γ has the form $\Gamma = \overline{X}, \overline{x}{:}\overline{T}, \overline{E}$ where free type variables in $\overline{T}$ and $\overline{E}$ are drawn from $\overline{X}$. We write $\cdot$ to denote the empty environment. Judgement forms are as follows:

- The formation judgement $\Gamma \vdash T\ ok$ states "in typing environment Γ, the type T is well-formed with respect to the class table and type variables declared in Γ".
- The formation judgement $\vdash \Gamma\ ok$ states that "the types in the environment are individually well-formed with respect to the class table and type variables in Γ".
- The novel type equivalence judgement $\Gamma \vdash E$ states that "the type equation E is a consequence of the conjoined equations in Γ".
- The mostly standard subtype judgement $\Gamma \vdash T <: U$ states that "type T is a subtype of U, given the equations in Γ".
- A typing judgment $\Gamma \vdash e : T$ states that "in the context of a typing environment Γ, the expression e has type T" with type variables in e and T drawn from Γ.
- A method well-formedness judgment $\vdash md\ ok\ in\ C\texttt{<}\overline{X}\texttt{>}$ states that "method definition md is valid in class $C\texttt{<}\overline{X}\texttt{>}$".
- A class well-formedness judgment $\vdash cd\ ok$ states that "class definition cd is valid".
- The judgement $e \rightarrow e$ states that "(closed) expression e reduces, in one step, to (closed) expression e'." As usual, the reduction relation is defined by both primitive *reduction* rules and contextual *evaluation* rules.

All of the judgment forms and helper definitions of Figures 15 and 16 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 $\vdash^{\mathcal{D}} cd\ ok$ for each class definition cd in $\mathcal{D}$ and the class hierarchy is a tree rooted at `object` (which we could easily formalize but do not).

The operation $mtype(T.m)$, 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 also computes the inherited constraints of m so it cannot simply return the syntactic signature of an intervening override but must examine its virtual definition.

The operation $mbody(T.m\texttt{<}\overline{T}\texttt{>})$, 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 $\overline{T}$.

The method formation judgements make use of the following definition:

Definition 1 (Satisfiable Equations). *An equation set $\overline{E} \equiv \overline{T}_1 = \overline{T}_2$, where $X \vdash \overline{T}_1, \overline{T}_2\ ok$, is* satisfiable*, written $\overline{X} \vdash \overline{E}$ satisfiable, if, and only if, there is some substitution $[\overline{U}/\overline{X}]$ such that $[\overline{U}/\overline{X}]\overline{T}_1 = [\overline{U}/\overline{X}]\overline{T}_2$ (syntactically).*

By extension, a type environment $\Gamma = \overline{X}, \overline{x}{:}\overline{T}, \overline{E}$ is satisfiable*, written $\vdash \Gamma$ satisfiable, if, and only if, $\overline{X} \vdash \overline{E}$ satisfiable.*

Note that it is possible to decide satisfiability using Robinson's first-order unification algorithm [3, 1].

```
using T; // The typed expressions namespace
public abstract class AnyTypedExp { }
public class TypedExp<T> : AnyTypedExp {
  Rep<T> rep; Exp<T> exp; public TypedExp(Rep<T> rep, Exp<T> exp) { this.rep = rep; this.exp = exp; }
}
namespace U { // Untyped expressions
  public abstract class Exp { ...
    public abstract AnyTypedExp TC();
  }
  public class Lit : Exp { ...
    public override AnyTypedExp TC()
      { return new TypedExp<int>(new IntRep(), new T.Lit(value)); }
  }
  public class Plus : Exp { ...
    public override AnyTypedExp TC() {
      switch (e1.TC(), e2.TC()) {
        case (TypedExp<A> te1, TypedExp<B> te2) :
          switch (te1.rep, te2.rep) {
            case (IntRep, IntRep) : return new TypedExp<int>(new IntRep(), new T.Plus(te1.exp, te2.exp));
            default : return null;
          }
        default : return null;
  } } }
  public class Cond : Exp { ...
    public override AnyTypedExp TC() {
      switch (e1.TC(), e2.TC(), e3.TC()) {
        case (TypedExp<A> te1, TypedExp<B> te2, TypedExp<C> te3) :
          switch (te1.rep) {
            case BoolRep :
              switch (Helper.IsEqual(te2.rep, te3.rep)) {
                 case Identical<D> : return new TypedExp<D>(te3.rep, new T.Cond<D>(te1.exp, te2.exp, te3.exp));
                 default : return null;
              }
            default : return null;
          }
        default : return null;
  } } }
  public class Tuple : Exp { ...
    public override AnyTypedExp TC() {
      switch (e1.TC(), e2.TC()) {
        case (TypedExp<A> te1, TypedExp<B> te2) :
          return new TypedExp<Pair<A,B>>(new PairRep<A,B>(te1.rep, te2.rep), new T.Tuple<A,B>(te1.exp, te2.exp));
        default : return null;
  } } }
  public class Fst : Exp { ...
    public override AnyTypedExp TC() {
      switch (e.TC()) {
        case (TypedExp<T> te) :
          switch (te.rep) {
            case PairRep<A,B> pairrep :return new TypedExp<A>(pairrep.fst, new T.Fst<A,B>(te.exp));
            default : return null;
          }
        default : return null;
  } } }
}
public abstract class Equal<A,B> { }
public class Identical<C> : Equal<C,C> { }
public class Helper {
  public static Equal<A,B> IsEqual<A,B>(Rep<A> r1, Rep<B> r2) {
    switch (r1,r2) {
      case (IntRep, IntRep) : return new Identical<int>();
      case (BoolRep, BoolRep) : return new Identical<bool>();
      case (PairRep<C,D> pairrep1, PairRep<E,F> pairrep2) :
        switch (IsEqual(pairrep1.fst, pairrep2.fst), IsEqual(pairrep1.snd,pairrep2.snd)) {
          case (Identical<G>, Identical<H>) : return new Identical<Pair<G,H>>();
          default : return null;
        }
    }
    return null;
  }
}
```

Figure 14: A GADT type-checker, recovering strongly-typed from untyped expressions (using `switch`)

Syntax:

(class def)	cd	::=	$\texttt{class}\ C\texttt{<}\overline{X}\texttt{>}\ \texttt{:}\ I\ \{\overline{T}\ \overline{f}\texttt{;}\ kd\ \overline{md}\}$
(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	::=	$x \mid e.f \mid e.m\texttt{<}\overline{T}\texttt{>(}\overline{e}\texttt{)} \mid \texttt{new}\ I\texttt{(}\overline{e}\texttt{)} \mid \texttt{(}T\texttt{)}e$
(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)		::=	$\texttt{<}\overline{X}\ \texttt{where}\ \overline{E}\ \texttt{>}\overline{T} \to T \quad (\overline{X}\ \texttt{is bound in}\ \overline{E}, \overline{T}, T)$
(simultaneous type and term substitutions)		::=	$[\overline{T}/\overline{X}], [\overline{e}/\overline{x}]$

Well-formed contexts and types:

$$\frac{\overline{X} \vdash \overline{T}\ , \overline{U}, \overline{V}\ \ ok}{\vdash \overline{X}, \overline{x} : \overline{T}\ , \overline{U}\texttt{=}\overline{V}\ \ ok} \qquad \frac{X \in \Gamma}{\Gamma \vdash X\ ok} \qquad \frac{\texttt{class}\ C\texttt{<}\overline{X}\texttt{>}\ \texttt{:}\ I\ \{\ldots\} \quad \Gamma \vdash \overline{T}\ ok \quad |\overline{T}| = |\overline{X}|}{\Gamma \vdash C\texttt{<}\overline{T}\texttt{>}\ ok}$$

Type Equivalence:

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

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

Subtyping:

$$(\text{sub-refl})\frac{\Gamma \vdash T\texttt{=}U}{\Gamma \vdash T <: U} \qquad \frac{\Gamma \vdash T <: U \quad \Gamma \vdash U <: V}{\Gamma \vdash T <: V} \qquad \frac{X \in \Gamma}{\Gamma \vdash X <: \texttt{object}} \qquad \frac{\mathcal{D}(C) = \texttt{class}\ C\texttt{<}\overline{X}\texttt{>}\ \texttt{:}\ I\ \{\ldots\} \quad \Gamma \vdash \overline{T}\ ok}{\Gamma \vdash C\texttt{<}\overline{T}\texttt{>} <: [\overline{T}/\overline{X}]I}$$

Typing:

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

$$(\text{ty-sub})\frac{\Gamma \vdash e : T \quad \Gamma \vdash T <: U}{\Gamma \vdash e : U} \qquad (\text{ty-new})\frac{\Gamma \vdash I\ ok \quad \mathit{fields}(I) = \overline{T}\ \overline{f} \quad \Gamma \vdash \overline{e} : \overline{T}}{\Gamma \vdash \texttt{new}\ I\texttt{(}\overline{e}\texttt{)} : I}$$

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

Method and Class Typing:

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

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

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

Figure 15: Syntax and typing rules for C$^\sharp$ minor

Operational Semantics:
(reduction rules)

$$\text{(r-fld)}\ \frac{\mathit{fields}(I) = \overline{T}\ \overline{f}}{\texttt{new}\ I\texttt{(}\overline{v}\texttt{)}\texttt{.}f_i \to v_i} \quad \text{(r-meth)}\ \frac{\mathit{mbody}(I.m\texttt{<}\overline{T}\texttt{>}) = \langle \overline{x}, e' \rangle}{\texttt{new}\ I\texttt{(}\overline{v}\texttt{)}\texttt{.}m\texttt{<}\overline{T}\texttt{>(}\overline{w}\texttt{)} \to [\overline{w}/\overline{x}, \texttt{new}\ I\texttt{(}\overline{v}\texttt{)/this}]e'} \quad \text{(r-cast)}\ \frac{\vdash I <: T}{\texttt{(}T\texttt{)new}\ I\texttt{(}\overline{v}\texttt{)} \to \texttt{new}\ I\texttt{(}\overline{v}\texttt{)}}$$

(evaluation rules)

$$\text{(c-new)}\ \frac{e \to e'}{\texttt{new}\ I\texttt{(}\overline{v}, e, \overline{e}\texttt{)} \to \texttt{new}\ I\texttt{(}\overline{v}, e', \overline{e}\texttt{)}} \quad \text{(c-fld)}\ \frac{e \to e'}{e\texttt{.}f_i \to e'\texttt{.}f_i} \quad \text{(c-cast)}\ \frac{e \to e'}{\texttt{(}T\texttt{)}\,e \to \texttt{(}T\texttt{)}\,e'}$$

$$\text{(c-meth-rcv)}\ \frac{e \to e'}{e\texttt{.}m\texttt{<}\overline{T}\texttt{>(}\overline{e}\texttt{)} \to e'\texttt{.}m\texttt{<}\overline{T}\texttt{>(}\overline{e}\texttt{)}} \quad \text{(c-meth-arg)}\ \frac{e \to e'}{v\texttt{.}m\texttt{<}\overline{T}\texttt{>(}\overline{v}, e, \overline{e}\texttt{)} \to v\texttt{.}m\texttt{<}\overline{T}\texttt{>(}\overline{v}, e', \overline{e}\texttt{)}}$$

Field lookup:

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

Method lookup:

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

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

Method dispatch:

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

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

Figure 16: Evaluation rules and helper definitions for C$^\sharp$ minor

Now some comments on the differences in our rules. Apart from the usual equivalence and congruence rules, the type equivalence judgement includes the novel hypothesis and *decomposition* rules, (eq-hyp) and (eq-decon), discussed at length in [3]: rule (eq-decon) states that class names are injective so that we can deduce the equivalence of corresponding type arguments from the equivalence of two instantiations of the same class. Full reflexivity is derivable from reflexivity on variables and the congruence rules. It is easy (and useful) to show that, in the context of no equations, $\overline{X} \vdash T\texttt{=}U$ implies $T = U$ (syntactically).

Cheney and Hinze also prove the following pragmatically useful result that relates equational reasoning in a satisfiable environment to unification (Proposition 1. of [3]):

THEOREM 1. *Assume* $\vdash \overline{X}, \overline{E}$ *ok and let* $\Theta \equiv [\overline{U}/\overline{X}]$ *be any* most-general unifier *of* $\overline{E}$*, then* $\overline{X}, \overline{E} \vdash T_1\texttt{=}T_2$ *if, and only if,* $\Theta(T_1) = \Theta(T_2)$ *(syntactically).*

If our hypothetical equations are satisfiable, then we can decide whether an equation is derivable by applying a most general unifier of the equational hypotheses and testing that the equated types are identical.

Our subtyping judgement $\Gamma \vdash T <: U$ is standard, except that the usual reflexivity rule is generalized by rule (sub-refl) to include equivalent (not just identical) types in the subtyping relation. Rule (ty-sub) can exploit these equations to assign more than one type to an expression (even if these types are otherwise unrelated by the class hierarchy).

Rule (ty-meth) imposes an additional premise: the actual, instantiated constraints of the method signature (if any) must be derivable from the equations in the context. In turn, rules (ok-virtual) and (ok-override) add the declared or inherited formal method constraints to the environment, before checking the method body: the body may assume the constraints hold, thus allowing more code to type-check.

Our type checking rules are not algorithmic in their current form. In particular, the rules do not give a strategy for proving equations or subtyping judgements and the type checking judgement for expressions is not syntax-directed because of rule (ty-sub). As a concession to producing an

algorithm, rules (ok-virtual) and (ok-override) require that the declared constraints in $\overline{E}$ are *satisfiable*. This ensures that an algorithm will only have to type check the bodies of methods that have (most general) solutions. This does not rule out any useful programs: methods with unsatisfiable constraints are effectively dead, since the preconditions for calling them can never be established. Rejecting unsatisfiable methods means that it should be easy to adapt an existing type checking algorithm for C$^\sharp$ minor without constraints: the modified algorithm first solves the equations to obtain a most general unifier, applies that unifier to the context, body and return type, and then proceeds with type checking as usual. Under the unifier, types that are equivalent must be syntactically identical (by Theorem 1), which makes subtyping and equivalence simple to test. At a method call, equational pre-conditions will either be syntactically true (all equated types are identical), or syntactically false (some equated types are distinct), signaling a type error.

Nevertheless, our proof of Type Soundness does *not* rely on the notion of *satisfiability*. Even if we define satisfiability to be vacuously true, type soundness still holds.

We now return to the proof (eliding to state standard lemmas like *Well-formedness*, *Weakening* and *Inversion*).

LEMMA 1 (SANITY). *Provided $\mathcal{D}$ is a valid class table:*

- *The relations $mtype(T.m) = \_$ and $fields(T) = \_$ define partial functions.*
- *If $\cdot \vdash T$ ok and $fields(T) = \overline{T}\ \overline{f}$ then $\overline{f}$ are disjoint and $\cdot \vdash \overline{T}$ ok.*
- *If $\cdot \vdash T$ ok and $mtype(T.m) = \texttt{<}\overline{X}\ \texttt{where}\ \overline{U_1}\texttt{=}\overline{U_2}\texttt{>}\overline{V} \to V$, then $\overline{X} \vdash \overline{U_1}, \overline{U_2}, \overline{V}, V$ ok.*

We prove the usual type and term substitution properties that follow, but a key lemma for our system is Lemma 4, that lets us discharge proven equational hypotheses from various judgement forms (a similar lemma appears in [3]).

LEMMA 2 (SUBSTITUTION PROPERTY FOR LOOKUP). *If $\mathcal{D}$ is a valid class table,*

- *If $fields(I) = \overline{T}\ \overline{f}$ then $fields([\overline{U}/\overline{Y}]I) = [\overline{U}/\overline{Y}]\overline{T}\ \overline{f}$.*
- *$mtype(I.m) = \texttt{<}\overline{X}\ \texttt{where}\ \overline{E}\texttt{>}\overline{T} \to T$ implies $mtype(([\overline{U}/\overline{Y}]I).m) = [\overline{U}/\overline{Y}](\texttt{<}\overline{X}\ \texttt{where}\ \overline{E}\texttt{>}\overline{T} \to T)$.*
- *$mtype(I.m)$ is undefined then $mtype(([\overline{U}/\overline{Y}]I).m)$ is undefined.*

LEMMA 3 (SUBSTITUTION FOR TYPES). *Let $\mathcal{J}$ range over all judgment forms, namely typing ($e : T$), type equivalence ($T\texttt{=}U$), type well-formedness (T ok) and subtyping ($T <: U$). If $\overline{X}, \overline{Y}, \overline{x}\texttt{:}\overline{T}, \overline{E} \vdash \mathcal{J}$ and $\overline{Y} \vdash \overline{U}$ ok then $\overline{Y}, \overline{x}\texttt{:}[\overline{U}/\overline{X}]\overline{T}, [\overline{U}/\overline{X}]\overline{E} \vdash [\overline{U}/\overline{X}]\mathcal{J}$.*

PROOF. Straightforward induction on the derivation of $\mathcal{J}$, using Lemma 2. □

LEMMA 4 (EQUATION ELIMINATION). *Let $\mathcal{J}$ range over typing, type equivalence and subtyping judgment forms ($e : T$, $T\texttt{=}U$ 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 $\mathcal{J}$. □

LEMMA 5 (SUBSTITUTION FOR TERMS). *If $\Gamma, \overline{x}\texttt{:}\overline{T} \vdash e : T$ and $\Gamma \vdash \overline{v} : \overline{T}$ then $\Gamma \vdash [\overline{v}/\overline{x}]e : T$.*

PROOF. By induction on the typing derivation. □

To prove Preservation we also need the following properties of inheritance:

LEMMA 6 (FIELD PRESERVATION). *Provided $\mathcal{D}$ is a valid class table, $\cdot \vdash T, U$ ok and $\cdot \vdash T <: U$, then $fields(U) = \overline{U}\ \overline{g}$ and $fields(T) = \overline{T}\ \overline{f}$ implies $T_i = U_i$ and $f_i = g_i$ for all $i \le |\overline{g}|$.*

LEMMA 7 (SIGNATURE PRESERVATION). *Provided $\mathcal{D}$ is a valid class table, $\cdot \vdash T, U$ ok and $\cdot \vdash T <: U$ then $mtype(U.m) = \texttt{<}\overline{X}\ \texttt{where}\ \overline{E}\texttt{>}\overline{V} \to V$ implies $mtype(T.m) = \texttt{<}\overline{X}\ \texttt{where}\ \overline{E}\texttt{>}\overline{V} \to V$.*

LEMMA 8 (SOUNDNESS FOR DISPATCH). *If $\mathcal{D}$ is a valid class table and $mbody(T.m\texttt{<}\overline{T}\texttt{>}) = \langle\overline{x}, e\rangle$ then, provided $\cdot \vdash T, \overline{T}$ ok and $mtype(T.m) = \texttt{<}\overline{X}\ \texttt{where}\ \overline{E}\texttt{>}\overline{U} \to U$ and $\cdot \vdash [\overline{T}/\overline{X}]\overline{E}$, there must be some type V such that $\cdot \vdash V$ ok, $\cdot \vdash T <: V$ and $\overline{x}\texttt{:}[\overline{T}/\overline{X}]\overline{U}, \texttt{this}\texttt{:}V \vdash e : [\overline{T}/\overline{X}]U$.*

PROOF. By induction on the relation $mbody(T.m\texttt{<}\overline{T}\texttt{>}) = \langle\overline{x}, \texttt{return}\ e\texttt{;}\rangle$ using Substitution Lemmas 3 and 4 and 2. □

THEOREM 2 (PRESERVATION). *If $\mathcal{D}$ is a valid class table and $\cdot \vdash e : T$ then $e \to e'$ implies $\cdot \vdash e : T$.*

PROOF. Assume $\mathcal{D}$ is a valid class table and prove

$$e \to e' \implies \forall T. \cdot \vdash e : T \implies \cdot \vdash e' : T$$

by induction on the reduction relation using Lemmas 5, 6, 7 and 8. □

The proof of Progress relies on Lemma 9. The lemma guarantees the presence of a dynamically resolved field or method body, given the existence of a member of the same name in a statically known superclass.

LEMMA 9 (RUNTIME LOOKUP). *Provided $\mathcal{D}$ is a valid class table, $\cdot \vdash T, U$ ok and and $\cdot \vdash T <: U$ then*

- *$fields(U) = \overline{U}\ \overline{g}$ implies $fields(T) = \overline{T}\ \overline{f}$, for some $\overline{T}, \overline{f}$, with $T_i = U_i$ and $f_i = g_i$ for all $i \le |\overline{g}|$.*
- *$mtype(U.m) = \texttt{<}\overline{X}\ \texttt{where}\ \overline{E}\texttt{>}\overline{V} \to V$ implies $mbody(T.m\texttt{<}\overline{T}\texttt{>}) = \langle\overline{x}, e\rangle$ for some $\overline{x}, e$ with $|\overline{x}| = |\overline{V}|$.*

To state the Progress Theorem in the presence of casts, as for FGJ, we first characterize the implicit *evaluation contexts*, $\mathcal{E}$, defined by the evaluation rules:

$$\begin{array}{rcl}
\mathcal{E} & ::= & [] \\
& | & \texttt{new}\ I(\overline{v}, \mathcal{E}, \overline{e}) \\
& | & \mathcal{E}.f \\
& | & \mathcal{E}.m\texttt{<}\overline{T}\texttt{>}(\overline{e}) \\
& | & v.m\texttt{<}\overline{T}\texttt{>}(\overline{v}, \mathcal{E}, \overline{e}) \\
& | & (T)\mathcal{E}
\end{array}$$

We define $\mathcal{E}[e]$ to be the obvious expression obtained by replacing the unique hole [] in $\mathcal{E}$ with e.

THEOREM 3 (PROGRESS). *If $\mathcal{D}$ is a valid class table and $\cdot \vdash e : T$ then:*

- *$e = v$ for some value v (e is fully evaluated), or*
- *$e \rightarrow e'$ for some e' (e can make progress), or*
- *$e = \mathcal{E}[\texttt{(}U\texttt{)new}\ I\ \texttt{(}\overline{v}\texttt{)}]$, for some evaluation context $\mathcal{E}$, types U and I and values $\overline{v}$ where $\neg \vdash I <: U$ (e is stuck, but only at the evaluation of a failed cast).*

PROOF. We assume $\mathcal{D}$ is a valid class table and show

$$\begin{array}{l}\Gamma \vdash e : T \implies \\ \quad \Gamma \vdash e : T \wedge \\ \quad \Gamma = \cdot \implies \\ \qquad (\exists v.e = v) \vee \\ \qquad (\exists e'.e \rightarrow e') \vee \\ \qquad (\exists \mathcal{E}, U, I, \overline{v}.e = \mathcal{E}[\texttt{(}U\texttt{)new}\ I\ \texttt{(}\overline{v}\texttt{)}] \wedge \neg \vdash I <: U)\end{array}$$

by induction on the typing relation, applying Lemma 9. □

THEOREM 4 (TYPE SOUNDNESS). *Define $e \rightarrow^\star e'$ to be the reflexive, transitive closure of $e \rightarrow e'$. If $\mathcal{D}$ is a valid class table and $\cdot \vdash e : T$, $e \rightarrow^\star e'$ with e' a normal form, then either e' is a value with $\cdot \vdash e' : T$, or a stuck expression of the form $\mathcal{E}[\texttt{(}U\texttt{)new}\ I\ \texttt{(}\overline{v}\texttt{)}]$ where $\neg \vdash I <: U$.*

PROOF. An easy induction over $e \rightarrow^\star e'$ using Theorems 3 and 2. □

5.1 Formalizing switch

We now show how to formalize a simple variant of the switch statement described informally in Section 3.4. Because C$^\sharp$ minor only has expressions, it is more convenient to formalize a switch *expression*, whose branches return a value, but the principles are similar. For brevity, we formalize a *unary* switch that dispatches on a single expression; n-ary switches are an easy exercise.

Figure 17 summarizes the additions. In the switch expression $\texttt{switch (}e_1\texttt{) \{}\overline{b}\ \texttt{default:}\ e_2\texttt{\}}$, we call e_1 the *scrutinee*, the possibly empty sequence $\overline{b}$ *branches*, and e_2 the *default* expression. A branch $\texttt{case}\ C\texttt{<}\overline{X}\texttt{>}\ x\texttt{:}\ e$ is a binding construct: the formal type parameters $\overline{X}$ of the *type pattern* $C\texttt{<}\overline{X}\texttt{>}$, are bound in the *branch expression* e; x is the *pattern variable*, of static type $C\texttt{<}\overline{X}\texttt{>}$.

The typing rule (ty-switch) states that the type of the scrutinee, $D\texttt{<}\overline{T}\texttt{>}$, is an instantiation, $\overline{T}$, of generic class D. For each branch, we check that the type pattern $C\texttt{<}\overline{X}\texttt{>}$ is well-formed and a generic subtype of some instantiation $\overline{U}$ of the scrutinee's generic class D, possibly mentioning type variables from $\overline{X}$. Since the environment is empty, but for $\overline{X}$, this amounts to walking the class-hierarchy from C, checking whether one of its superclasses is defined by specializing D at $\overline{U}$. In the Haskell type system for case expressions, this step corresponds to looking up the declared, formal result type of a constructor in a datatype: but in the OO setting, we need to additionally traverse the class hierarchy. We then equate the instantiations $\overline{T}$ and $\overline{U}$ adding the (necessarily) fresh type parameters $\overline{X}$ to the environment. Provided the resulting environment is satisfiable, we then check that the type of the branch under this assumption and the binding of the scrutinee, as variable x at the pattern type, is U. If the environment is unsatisfiable, the branch is dead and need not be checked. All branch expressions, including the default expression, must have the same result type U, but this may vary with the different equational hypotheses established by each branch. The reduction semantics ensures that the branch expression will only be evaluated when $\overline{T}$ and $\overline{U}$ are, in fact equal, for some instantiation of $\overline{X}$, i.e. when the runtime type of the scrutinee derives from C. Because the type of the branch expression must be generic in the hypothetical types $\overline{X}$, what this instantiation actually is, at runtime, cannot affect the type of the branch, beyond what is provable from the equational constraints. The type of the entire switch expression is U. Since this coincides with the type of the default expression, it must be well-formed in the original environment, Γ. In particular, the result type cannot mention any "existential" type variables introduced by the branches.

Fortunately, the reduction rules are much easier to explain. Rule (c-switch) steps the evaluation of the scrutinee, if it is not a value. Rule (r-succeed) reduces the entire switch to the first branch expression, provided the runtime type of the scrutinee's value derives from the pattern type. The runtime instantiation $\overline{T}$ of C is used to instantiate the type parameters of the branch; the scrutinee value replaces the pattern variable. Rule (r-fail) discards the first branch of the switch, when the type of the scrutinee does not derive from C. Rule (r-default) returns the default value, when the list of branches has been exhausted. So, if the scrutinee evaluates at all, the branches are tested in sequence, until either one is taken, or we proceed with the default.

Provided we extend the definition of evaluation contexts of Section 5 to cater for switch expressions,

$$\mathcal{E} \quad ::= \quad \ldots \mid \texttt{switch (}\mathcal{E}\texttt{) \{}\overline{b}\ \texttt{default:}\ e\texttt{\}}$$

then we can again re-establish:

THEOREM 5 (TYPE SOUNDNESS). *C$^\sharp$ minor, extended with equational constraints and switch expressions, remains sound, in the sense of Theorem 4.*

Again, the satisfiability requirement is only imposed as a concession to producing a type checking *algorithm* that need not attempt to type statically dead branches under nonsensical equations. But why do we predicate the typeability of a branch on satisfiability of the environment, rather than requiring that the environment is satisfiable *and* the branch expression typeable (*i.e.* is the implication necessary)? The answer is that to do otherwise would break the type substitution property required to prove Preservation. Although satisfiability itself is closed under substitution, substituting into a proper subset of a set of satisfiable equations can break the satisfiability of the entire set. To accommodate the effects of reduction, in which branches that were previously live (satisfiable), can become dead by type substitution, the type system needs to tolerate dead branches. In turn, this leniency ensures that intermediate stages of reduction remain typeable. A possible alternative, also discussed by [3], might be to redefine type substitution on expressions to prune dead branches, but that seems like a radical departure from tradition. The problem does not arise for methods, because reduction does not inline the definition of a method until it is reduced.

6. RELATED WORK

Generalized algebraic data types were introduced by Xi, Chen and Chen under the name 'guarded recursive datatype

Syntax:

$$\begin{array}{rrcl} \text{(expression)} & e & ::= & \ldots \mid \texttt{switch}\ (e)\ \{\overline{b}\ \texttt{default:}\ e'\} \\ \text{(branch)} & b & ::= & \texttt{case}\ C\texttt{<}\overline{X}\texttt{>}\ x\texttt{:}\ e \end{array}$$

Typing:

$$\text{(ty-switch)}\ \frac{\begin{array}{l}\Gamma \vdash e_1 : D\texttt{<}\overline{T}\texttt{>} \qquad \Gamma \vdash e_2 : U \\ \forall(\texttt{case}\ C\texttt{<}\overline{X}\texttt{>}\ x\texttt{:}\ e) \in \overline{b}.\quad \overline{X} \vdash C\texttt{<}\overline{X}\texttt{>}\ ok \quad \overline{X} \vdash C\texttt{<}\overline{X}\texttt{>} <: D\texttt{<}\overline{U}\texttt{>} \quad \overline{X} \notin \Gamma \\ \qquad\qquad (\vdash \Gamma, \overline{X}, \overline{T}\texttt{=}\overline{U}\ \textit{satisfiable} \implies \Gamma, \overline{X}, \overline{T}\texttt{=}\overline{U}, x\texttt{:}C\texttt{<}\overline{X}\texttt{>} \vdash e : U)\end{array}}{\Gamma \vdash \texttt{switch}\ (e_1)\ \{\overline{b}\ \texttt{default:}\ e_2\} : U}$$

Operational Semantics:

$$\text{(r-succeed)}\ \frac{\vdash I <: C\texttt{<}\overline{T}\texttt{>}}{\texttt{switch}\ (\texttt{new}\ I(\overline{v}))\ \{\texttt{case}\ C\texttt{<}\overline{X}\texttt{>}\ x\texttt{:}\ e_1\,\overline{b}\ \texttt{default:}\ e_2\} \to [\texttt{new}\ I(\overline{v})/x, \overline{T}/\overline{X}]e_1}$$

$$\text{(r-fail)}\ \frac{\nexists \overline{T}.\ \vdash I <: C\texttt{<}\overline{T}\texttt{>}}{\texttt{switch}\ (\texttt{new}\ I(\overline{v}))\ \{\texttt{case}\ C\texttt{<}\overline{X}\texttt{>}\ x\texttt{:}\ e_1\,\overline{b}\ \texttt{default:}\ e_2\} \to \texttt{switch}\ (\texttt{new}\ I(\overline{v}))\ \{\overline{b}\ \texttt{default:}\ e_2\}}$$

$$\text{(r-default)}\ \frac{|\overline{b}| = 0}{\texttt{switch}\ (v)\ \{\overline{b}\ \texttt{default:}\ e\} \to e} \qquad \text{(c-switch)}\ \frac{e_1 \to e_1'}{\texttt{switch}\ (e_1)\ \{\overline{b}\ \texttt{default:}\ e_2\} \to \texttt{switch}\ (e_1')\ \{\overline{b}\ \texttt{default:}\ e_2\}}$$

Figure 17: Extensions to C♯ minor for switch

constructors' [25], and were conceived as a means of representing types at runtime [5]. Independently, Cheney and Hinze formulated a very similar system, which they call 'first-class phantom types' [3]. Our formal characterization of type equality is similar to theirs; in particular, the use of a decomposition rule. In contrast, Xi, Chen and Chen use a semantic notion of entailment, and then show that a rule-based characterization of the mixed-prefix unification algorithm is sound and complete with respect to the semantics. Hinze's subsequent tutorial article on phantom types [8] is a rich source of examples.

Type inference for GADTs has received some attention recently. Simonet and Pottier cast it in the framework of the HM(X) constraint-based generalization of Hindley-Milner type inference for ML [23], and show that when 'X' is type equality, with suitable annotations on **case** constructs, type inference reduces to a decidable constraint solving problem. Peyton Jones, Washburn and Weirich take a different tack, solving unification-like problems on-the-fly. Their type system uses most general unifiers, avoiding equations [17].

GADTs can be seen as a step towards full-blown dependent types in programming. Sheard explores this connection with his recent programming language Omega [22, 21], using GADTs to express strong invariants over data structures such as binomial heaps and red-black trees.

Until now there has been no work on GADTs in the context of object-oriented programming. Sestoft noticed that C♯ generics can be used to express typed expressions and evaluation [19]; independently Meijer [12] presented a C♯ translation of the typed expression example from [17].

There have been a number of proposals to extend OO languages like Java with (non-generalized) algebraic datatypes [14, 26, 13]. Some of these proposals support the features (2: non-regular recursion) and (3: existential type parameters) identified in Section 2.2, but restrict 'case' classes to take the same type parameters as their superclass.

In this paper we described how the existing generics capability of C♯ and Java can be used to write only a limited class of *functions* expressed over GADTs, namely those that are 'generic' in the type parameters of the GADT. A similar restriction applied to *inductive* types in an early version of the Coq proof assistant [15]. Its elimination rule for inductive types used substitution on types, just as the rule for subclassing and signature refinement does in C♯ and Java. More recent versions of Coq support a more expressive pattern matching construct that uses unification to solve type equalities [4], providing similar power to our proposal for equational constraints, but in a much richer setting.

7. CONCLUSION AND FUTURE WORK

We have shown how the combination of generics, subclassing, and virtual dispatch supports the definition and limited use of generalized algebraic data types in object-oriented programming languages. To achieve full expressivity for GADTs in C♯ we have proposed the addition of equational constraints to the language; this extension also makes it much easier to code certain operations over ordinary algebraic datatypes such as `Flatten` and `Unzip` on lists. For complex examples, virtual dispatch is impractical, and so for convenience we have proposed a generalization of the `switch` statement. Although we have not discussed this, our extensions have a natural *erasure* semantics, making them compatible with the erasure-based generics of Java as well as the runtime-type passing semantics of C♯.

We plan to implement our extensions in a compiler for C♯ version 2.0; we do not foresee any difficult interactions with features from the complete language.

Future work includes studying formal translations from System F with GADTs, into C♯ minor, with and without equational constraints, and extending the previous translation of Kennedy and Syme [10]. We have begun looking at

two variants of System F: one, with full support for GADTs, in the style of [25, 18, 17, 3], and utilizing equational constraints in the elimination rule for **case** (*strong-case*); the other, without equations, but with a **case**-elimination rule based on substitution (*weak-case*). The latter system characterizes precisely the GADT-manipulating programs which can be written in unextended C♯ without the use of runtime casts. Interestingly, the use of type-decomposition in deriving equation judgments turns out to be crucial: given a term that makes use of equations and the *strong-case* rule, but containing no use of decomposition, there exists an equivalent term derivable in the *weak-case* system without the use of equations at all.

We are also investigating generalizing constraints still further, specifying arbitrary subtype constraints on methods. This could subsume upper bounds, lower bounds, and equations. In particular, we would like the equivalence induced by subtyping ($T \cong U$ iff $T <: U$ and $U <: T$) to coincide with our equivalence relation as axiomatized in Figure 15. As here, the use of a decomposition rule is crucial: from $C\texttt{<}T\texttt{>} <: C\texttt{<}U\texttt{>}$ we deduce that $T <: U$ and $U <: T$, as classes behave invariantly with respect to subtyping.

Acknowledgements

Thanks to Nick Benton, Brian Grunkemeyer, Gavin Bierman, Simon Peyton Jones and Don Syme for valuable discussions.

8. REFERENCES

[1] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[2] K. B. Bruce, L. Cardelli, G. Castagna, J. Eifrig, S. F. Smith, V. Trifonov, G. T. Leavens, and B. C. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.

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

[4] T. Coquand. Pattern matching with dependent types. In *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, 1992.

[5] K. Crary, S. Weirich, and G. Morrisett. Intensional polymorphism in type erasure semantics. In *Journal of Functional Programming*, November 2002.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

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

[8] R. Hinze. Fun with phantom types. In J. Gibbons and O. de Moor, editors, *The Fun of Programming*, pages 245–262. Palgrave Macmillan, Mar. 2003.

[9] 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.

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

[11] 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.

[12] E. Meijer, November 2004. Presentation to IFIP WG 2.8 (private communication).

[13] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. The Scala language specification, 2005. Available from `http://scala.epfl.ch/`.

[14] M. Odersky and P. Wadler. Pizza into Java: Translating Theory into Practice. In *24th ACM Symposium on Principles of Programming Languages (POPL)*, pages 146–159, 1997.

[15] C. Paulin-Mohring. Inductive definitions in the system Coq: Rules and properties. Technical Report 92-49, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, December 1992.

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

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

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

[19] P. Sestoft. Representing typed expressions, December 2001. Code sample in `http://www.dina.kvl.dk/~sestoft/gcsharp/#expr`.

[20] P. Sestoft and H. I. Hansen. *C# Precisely*. MIT Press, October 2004.

[21] T. Sheard. Languages of the future. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 116–119. ACM Press, 2004.

[22] T. Sheard and E. Pasalic. Meta-programming with built-in type equality. In *Fourth International Workshop on Logical Frameworks and Meta-languages (LFM'04)*, July 2004.

[23] V. Simonet and F. Pottier. Constraint-based type inference for guarded algebraic data types. Research Report 5462, INRIA, Jan. 2005.

[24] S. 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`).

[25] 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.

[26] M. Zenger and M. Odersky. Extensible algebraic datatypes with defaults. In *ICFP '01: Proceedings of the 6th ACM International Conference on Functional Programming*, pages 241–252. ACM Press, 2001.