System F with Type Equality Coercions

Martin Sulzmann
National University of Singapore
Email: sulzmann@comp.nus.edu.sg

Manuel M. T. Chakravarty
Computer Science & Engineering
University of New South Wales
Email: chak@cse.unsw.edu.au

Simon Peyton Jones   Kevin Donnelly
Microsoft Research Ltd
Cambridge, England
Email: {simonpj,t-kevind}@microsoft.com

UNSW-CSE-TR-0624
November 2006

THE UNIVERSITY OF
NEW SOUTH WALES

School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia

Abstract

We introduce System F_C, which extends System F with support for non-syntactic type equality. There are two main extensions: (i) explicit witnesses for type equalities, and (ii) open, non-parametric type functions, given meaning by top-level equality axioms. Unlike System F, F_C is expressive enough to serve as a target for several different source-language features, including Haskell's `newtype`, generalised algebraic data types, associated types, functional dependencies, and perhaps more besides.

System F with Type Equality Coercions

Martin Sulzmann
School of Computing
National University of Singapore
sulzmann@comp.nus.edu.sg

Manuel M. T. Chakravarty
Computer Science & Engineering
University of New South Wales
chak@cse.unsw.edu.au

Simon Peyton Jones, Kevin Donnelly
Microsoft Research Ltd
Cambridge, England
{simonpj,t-kevind}@microsoft.com

Abstract

We introduce System F_C, which extends System F with support for non-syntactic type equality. There are two main extensions: (i) explicit witnesses for type equalities, and (ii) open, non-parametric type functions, given meaning by top-level equality axioms. Unlike System F, F_C is expressive enough to serve as a target for several different source-language features, including Haskell's `newtype`, generalised algebraic data types, associated types, functional dependencies, and perhaps more besides.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure

General Terms Languages, Theory

Keywords Typed intermediate language, advanced type features

1. Introduction

The polymorphic lambda calculus, System F, is popular as a highly-expressive typed intermediate language in compilers for functional languages. However, language designers have begun to experiment with a variety of type systems that are difficult or impossible to translate into System F, such as functional dependencies [21], generalised algebraic data types (GADTs) [44, 31], and associated types [6, 5]. For example, when we added GADTs to GHC, we extended GHC's intermediate language with GADTs as well, even though GADTs are arguably an over-sophisticated addition to a typed intermediate language. But when it came to associated types, even with this richer intermediate language, the translation became extremely clumsy or in places impossible.

In this paper we resolve this problem by presenting System $F_C(X)$, a super-set of F that is both *more foundational* and *more powerful* than adding *ad hoc* extensions to System F such as GADTs or associated types. $F_C(X)$ uses explicit type-equality coercions as witnesses to justify explicit type-cast operations. Like types, coercions are erased before running the program, so they are guaranteed to have no run-time cost.

This single mechanism allows a very direct encoding of associated types and GADTs, and allows us to deal with some exotic functional-dependency programs that GHC currently rejects on the grounds that they have no System-F translation (§2). Our specific contributions are these:

Abridged version appears in *The Third ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI'07)*, January 16, 2007, Nice, France, ACM Press.

- We give a formal description of System F_C, our new intermediate language, including its type system, operational semantics, soundness result, and erasure properties (§3). There are two distinct extensions. The first, explicit equality witnesses, gives a system equivalent in power to System F + GADTs (§3.2); the second introduces non-parametric type functions, and adds substantial new power, well beyond System F + GADTs (§3.3).
- A distinctive property of F_C's type functions is that they are *open* (§3.4). Here we use "open" in the same sense that Haskell type classes are open: just as a newly defined type can be made an instance of an existing class, so in F_C we can extend an existing type function with a case for the new type. This property is crucial to the translation of associated types.
- The system is very general, and its soundness requires that the axioms stated as part of the program text are *consistent* (§3.5). That is why we call the system $F_C(X)$: the "X" indicates that it is parametrised over a decision procedure for checking consistency, rather than baking in a particular decision procedure. (We often omit the "(X)" for brevity.) Conditions identified in earlier work on GADTs, associated types, and functional dependencies, already define such decision procedures.
- A major goal is that F_C should be a *practical* compiler intermediate language. We have paid particular attention to ensuring that F_C programs are robust to program transformation (§3.8).
- It must obviously be *possible* to translate the source language into the intermediate language; but it is also highly desirable that it be *straightforward*. We demonstrate that F_C has this property, by sketching a type-preserving translation of two source language idioms, namely GADTs (Section 4) and associated types (Section 5). The latter, and the corresponding translation for functional dependencies, are more general than all previous type-preserving translations for these features.

System F_C has no new foundational content: rather, it is an intriguing and practically-useful application of techniques that have been well studied in the type-theory community. Several other calculi exist that might in principle be used for our purpose, but they generally do not handle open type functions, are less robust to transformation, and are significantly more complicated. We defer a comparison with related work until §6.

To substantiate our claim that F_C is practical, we have implemented it in GHC, a state-of-the-art compiler for Haskell, including both GADTs and associated (data) types. This is not just a prototype; F_C now *is* GHC's intermediate language.

F_C does not strive to do everything; rather we hope that it strikes an elegant balance between expressiveness and complexity. While our motivating examples were GADTs and associated types, we believe that F_C may have much wider application as a typed target for sophisticated HOT (higher-order typed) source languages.

2. The key ideas

No compiler uses *pure* System F as an intermediate language, because some source-language constructs can only be desugared into pure System F by very heavy encodings. A good example is the algebraic data types of Haskell or ML, which are made more complicated in Haskell because algebraic data types can capture existential type variables. To avoid heavy encoding, most compilers invariably extend System F by adding algebraic data types, data constructors, and **case** expressions. We will use F_A to describe System F extended in this way, where the data constructors are allowed to have existential components [24], type variables can be of higher kind, and type constructor applications can be partial.

Over the last few years, source languages (notably Haskell) have started to explore language features that embody *non-syntactic* or *definitional* type equality. These features include functional dependencies [16], generalised algebraic data types (GADTs) [44, 37], and associated types [6, 5]. All three are difficult or impossible to translate into System F — and yet the alternative of simply extending System F by adding functional dependencies, GADTs, and associated types, seems wildly unattractive. Where would one stop?

In the rest of this section we informally present System F_C, an extension of System F that resolves the dilemma. We show how it can serve as a target for each of the three examples. The formal details are presented in §3. Throughout we use `typewriter font` for source-code, and *italics* for F_C.

2.1 GADTs

Consider the following simple type-safe evaluator, often used as the poster child of GADTs, written in the GADT extension of Haskell supported by GHC:

```
data Exp a where
  Zero :: Exp Int
  Succ :: Exp Int -> Exp Int
  Pair :: Exp b -> Exp c -> Exp (b, c)

eval :: Exp a -> a
eval Zero       = 0
eval (Succ e)   = eval e + 1
eval (Pair x y) = (eval x, eval y)

main = eval (Pair (Succ Zero) Zero)
```

The key point about this program, and the aspect that is hard to express in System F, is that in the `Zero` branch of `eval`, the type variable `a` is the same as `Int`, even though the two are syntactically quite different. That is why the `0` in the `Zero` branch is well-typed in a context expecting a result of type `a`.

Rather than extend the intermediate language with GADTs themselves — GHC's pre-F_C "solution" — we instead propose a general mechanism of parameterising functions with *type equalities*, written $\sigma_1 \sim \sigma_2$, witnessed by *coercions*. Coercion types are passed around using System F's existing type passing facilities and enable representing GADTs by ordinary algebraic data types encapsulating such type equality coercions.

Specifically, we translate the GADT `Exp` to an ordinary algebraic data type, where each variant is parametrised by a coercion:

$$\begin{array}{l}\textbf{data}\ \mathit{Exp}\ :\ \star \to \star\ \textbf{where}\\ \quad \mathit{Zero}\ :\ \forall a.\,(a \sim \mathit{Int})\ \Rightarrow\ \mathit{Exp}\ a\\ \quad \mathit{Succ}\ :\ \forall a.\,(a \sim \mathit{Int})\ \Rightarrow\ \mathit{Exp}\ \mathit{Int}\ \to\ \mathit{Exp}\ a\\ \quad \mathit{Pair}\ :\ \forall abc.\,(a \sim (b,c))\ \Rightarrow\ \mathit{Exp}\ b\ \to\ \mathit{Exp}\ c\ \to\ \mathit{Exp}\ a\end{array}$$

So far, this is quite standard; indeed, several authors present GADTs in the source language using a syntax involving explicit equality constraints, similar to that above [44, 10]. However, for us the equality constraints are extra type arguments to the constructor, which must be given when the constructor is applied, and which are brought into scope by pattern matching. The "⇒" is syntactic sugar, and we sloppily omitted the kind of the quantified type variables, so the type of Zero is really this:

$$\mathit{Zero}\ :\ \forall\ a{:}\star.\ \forall(\mathit{co}{:}a \sim \mathit{Int}).\ \mathit{Exp}\ a$$

Here a ranges over *types*, of kind $\star$, while co ranges over *coercions*, of kind $a \sim \mathit{Int}$. An important property of our approach is that *coercions are types*, and hence, *equalities* $\tau_1 \sim \tau_2$ *are kinds*. An equality kind $\tau_1 \sim \tau_2$ categorises all coercion types that witness the interchangeability of the two types τ_1 and τ_2. So, our slogan is *propositions as kinds*, and *proofs as (coercion) types.*

Coercion types may be formed from a set of elementary coercions that correspond to the rules of equational logic; for example, $\mathit{Int}\ : (\mathit{Int} \sim \mathit{Int})$ is an instance of the reflexivity of equality and $\mathrm{sym}\ \mathit{co} : (\mathit{Int} \sim a)$, with $\mathit{co} : (a \sim \mathit{Int})$, is an instance of symmetry. A call of the constructor Zero must be given a type (to instantiate a) and a coercion (to instantiate co), thus for example:

$$\mathit{Zero}\ \mathit{Int}\ \mathit{Int}\ :\ \mathit{Exp}\ \mathit{Int}$$

As indicated above, regular types like Int, when interpreted as coercions, witness reflexivity.

Just like value arguments, the coercions passed to a constructor when it is built are made available again by pattern matching. Here, then, is the code of `eval` in F_C:

$$\begin{array}{l}\mathit{eval}\ =\ \Lambda a{:}\star.\lambda e{:}\mathit{Exp}\ a.\\ \quad \textbf{case}\ e\ \textbf{of}\\ \qquad \mathit{Zero}\ (\mathit{co}{:}a \sim \mathit{Int})\ \to\\ \qquad\quad 0 \blacktriangleright \mathrm{sym}\ \mathit{co}\\ \qquad \mathit{Succ}\ (\mathit{co}{:}a \sim \mathit{Int})\ (e'{:}\mathit{Exp}\ \mathit{Int})\ \to\\ \qquad\quad (\mathit{eval}\ \mathit{Int}\ e'\ +\ 1) \blacktriangleright \mathrm{sym}\ \mathit{co}\\ \qquad \mathit{Pair}\ (b{:}\star)\ (c{:}\star)\ (\mathit{co}{:}a \sim (b,\ c))\\ \qquad\quad (e_1{:}\mathit{Exp}\ b)\ (e_2{:}\mathit{Exp}\ c)\ \to\\ \qquad\quad (\mathit{eval}\ b\ e_1,\ \mathit{eval}\ c\ e_2) \blacktriangleright \mathrm{sym}\ \mathit{co}\end{array}$$

The form $\Lambda a{:}\star.e$ abstracts over types, as usual. In the first alternative of the **case** expression, the pattern binds the coercion type argument of Zero to co. We use the symmetry of equality in $(\mathrm{sym}\ \mathit{co})$ to get a coercion from Int to a and use that to cast the type of 0 to a, using the *cast expression* $0 \blacktriangleright \mathrm{sym}\ \mathit{co}$. Cast expressions have no *operational* effect, but they serve to explain to the type system when a value of one type (here Int) should be treated as another (here a), and provide evidence for this equivalence. In general, the form $e \blacktriangleright g$ has type t_2 if $e\ : t_1$ and $g\ :\ (t_1 \sim t_2)$. So, $\mathit{eval}\ \mathit{Int}\ (\mathit{Zero}\ \mathit{Int}\ \mathit{Int}))$ is of type Int as required by eval's signature. We shall discuss coercion types and their kinds in more detail in §3.2.

In a similar manner, the recently-proposed extended algebraic data types [41], which add equality and predicate constraints to GADTs, can be translated to F_C.

2.2 Associated types

Associated types are a recently-proposed extension to Haskell's type-class mechanism [6, 5]. They offer open, type-indexed types that are associated with a type class. Here is a standard example:

```
class Collects c where
  type Elem c      -- associated type synonym
  empty  :: c
  insert :: Elem c -> c -> c
```

The class `Collects` abstracts over a family of containers, where the representation type of the container, `c`, defines (or constrains) the type of its elements `Elem c`. That is, `Elem` is a type-level function that transforms the collection type to the element type. Just as `insert` is non-parametric – its implementation varies depending on `c` – so is `Elem`. For example, a list container can contain elements of any type supporting equality, and a bit-set container might represent a collection of characters:

```
instance Eq e => Collects [e] where
    {type Elem [e] = e; ...}
instance Collects BitSet where
    {type Elem BitSet = Char; ...}
```

Generally, type classes are translated into System F [17] by (1) turning each class into a record type, called a dictionary, containing the class methods, (2) converting each instance into a dictionary value, and (3) passing such dictionaries to whichever function mentions a class in its signature. For example, a function of type `negate :: Num a => a -> a` will translate to $negate : NumDict\ a \rightarrow a \rightarrow a$, where $NumDict$ is the record generated from the class `Num`.

A record only encapsulates values, so what to do about associated types, such as `Elem` in the example? The system given in [6] translates each associated type into an additional type parameter of the class's dictionary type, provided the class and instance declarations abide by some moderate constraints [6]. For example, the class `Collects` translates to dictionary type $CollectsDict\ c\ e$, where e represents $Elem\ c$ and where all occurrences of $Elem\ c$ of the method signatures have been replaced by the new type parameter e. So, the (System F) type for $insert$ would now be $CollectDict\ c\ e \rightarrow e \rightarrow c \rightarrow c$. The required type transformations become more complex when associated types occur in data types; the data types have to be rewritten substantially during translation, which can be a considerable burden in a compiler.

Type equality coercions enable a far more direct translation. Here is the translation of `Collects` into $\mathrm{F_C}$:

$$
\begin{array}{l}
\mathbf{type}\ Elem\ :\ \star \rightarrow \star \\
\mathbf{data}\ CollectsDict\ c\ = \\
\quad Collects\ \{empty\ :\ c;\ insert\ :\ Elem\ c\ \rightarrow\ c\ \rightarrow\ c\}
\end{array}
$$

The dictionary type is as in a translation without associated types. The **type** declaration in $\mathrm{F_C}$ introduces a new *type function*. An instance declaration for `Collects` is translated to (a) a dictionary transformer for the values and (b) an equality axiom that describes (part) of the interpretation for the type function $Elem$. For example, here is the translation into $\mathrm{F_C}$ of the `Collects Bitset` instance:

$$
\begin{array}{l}
\mathbf{axiom}\ elemBS\ :\ Elem\ BitSet \sim Char \\
dCollectsBS\ :\ CollectsDict\ Bitset \\
dCollectsBS\ =\ ...
\end{array}
$$

The **axiom** definition introduces a new, named *coercion constant*, $elemBS$, which serves as a witness of the equality asserted by the axiom; here, that we can convert between types of the form $Elem\ BitSet$ and $Char$. Using this coercion, we can $insert$ the character 'b' into a $BitSet$ by applying the coercion $elemBS$ backwards to 'b', thus:

$$(\text{'b'} \blacktriangleright (\mathrm{sym}\ elemBS)) : Elem\ BitSet$$

This argument fits the signature of $insert$.

In short, System $\mathrm{F_C}$ supports a very direct translation of associated types, in contrast to the clumsy one described in [6]. What is more, there are several obvious extensions to the latter paper that cannot be translated into System F at all, even clumsily, and $\mathrm{F_C}$ supports them too, as we sketch in Section 5.

2.3 Functional dependencies

Functional dependencies are another popular extension of Haskell's type-class mechanism [21]. With functional dependencies, we can encode a function over types `F` as a relation, thus

```
class F a b | a -> b
instance F Int Bool
```

However, some programs involving functional dependencies are impossible to translate into System F. For example, a useful idiom in type-level programming is to abstract over the co-domain of a type function by way of an existential type, the `b` in this example:

```
data T a = forall b. F a b => MkT (b -> b)
```

In this Haskell declaration, `MkT` is the constructor of type `T`, capturing an existential type variable `b`. One might hope that the following function would type-check:

```
combine :: T a -> T a -> T a
combine (MkT f) (MkT f') = MkT (f . f')
```

After all, since the type `a` functionally determines `b`, `f` and `f'` must have the same type. Yet GHC rejects this program, because it cannot be translated into System $\mathrm{F_A}$, because `f` and `f'` each have distinct, existentially-quantified types, and there is no way to express their (non-syntactic) identity in $\mathrm{F_A}$.

It is easy to translate this example into $\mathrm{F_C}$, however:

$$
\begin{array}{l}
\mathbf{type}\ F1\ :\ \star \rightarrow \star \\
\mathbf{data}\ FDict\ :\ \star \rightarrow \star \rightarrow \star\ \mathbf{where} \\
\quad F\ :\ \forall a\ b.\,(b \sim F1\ a)\ \Rightarrow\ FDict\ a\ b \\
\mathbf{axiom}\ fIntBool\ :\ F1\ Int \sim Bool \\
\mathbf{data}\ T : \star \rightarrow \star\ \mathbf{where} \\
\quad MkT\ :\ \forall a\ b.FDict\ a\ b\ \rightarrow\ (b\ \rightarrow\ b)\ \rightarrow\ T\ a \\
\\
combine\ :\ T\ a\ \rightarrow\ T\ a\ \rightarrow\ T\ a \\
combine\ (MkT\ b\ \ (F\ (co\ :\ b \sim F1\ a))\ f) \\
\qquad\quad (MkT\ b'\ \ (F\ (co' : b' \sim F1\ a))\ f') \\
\quad =\ MkT\ a\ b\ (F\ a\ b\ co)\ (f\ .\ (f' \blacktriangleright d_2)) \\
\quad \mathbf{where} \\
\qquad d_1\ :\ (b' \sim b) \qquad\qquad\quad =\ co' \circ \mathrm{sym}\ co \\
\qquad d_2\ :\ (b'\ \rightarrow\ b' \sim b\ \rightarrow\ b) =\ d_1\ \rightarrow\ d_1
\end{array}
$$

The functional dependency is expressed as a type function $F1$, with one equality axiom per instance. (In general there might be many functional dependencies for a single class.) The dictionary for class `F` includes a witness that indeed b is equal to $F1\ a$, as you can see from the declaration of constructor F. When pattern matching in $combine$, we gain access to these witnesses, and can use them to cast f' so that it has the same type as f. (To construct the witness d_1 we use the coercion combinators $\mathrm{sym}\,\cdot$ and $\cdot \circ \cdot$, which represent symmetry and transitivity; and from d_1 we build the witness d_2.)

Even in the absence of existential types, there are reasonable source programs involving functional dependencies that have no System F translation, and hence are rejected by GHC. We have encountered this problem in real programs, but here is a boiled-down example, using the same class `F` as before:

```
class D a where { op :: F a b => a -> b }
instance D Int where { op _ = True }
```

The crucial point is that the context `F a b` of the signature of `op` constrains the parameter of the enclosing type class `D`. This becomes a problem when typing the definition of `op` in the instance `D Int`. In `D`'s dictionary $DDict$, we have $op\ :\ \forall b.C\ a\ b\ \rightarrow\ a\ \rightarrow b$ with b universally quantified, but in the instance declaration, we would need to instantiate b with $Bool$. The instance declaration for `D` cannot be translated into System F. Using $\mathrm{F_C}$, this problem is easily solved: the coercion in the dictionary for `F` enables the result of `op` to be cast to type b as required.

To summarise, a compiler that uses translation into System F (or $\mathrm{F_A}$) must reject some reasonable (albeit slightly exotic) programs involving functional dependencies, and also similar programs involving associated types. The extra expressiveness of System $\mathrm{F_C}$ solves the problem neatly.

2.4 Translating `newtype`

F_C is extremely expressive, and can support language features beyond those we have discussed so far. Another example are Haskell 98's `newtype` declarations:

```
newtype T = MkT (T -> T)
```

In Haskell, this declares `T` to be isomorphic to `T->T`, but there is no good way to express that in System F. In the past, GHC has handled this with an *ad hoc* hack, but F_C allows it to be handled directly, by introducing a new axiom

$$\textbf{axiom}\ CoT\ :\ (T \to T) \sim T$$

2.5 Summary

In this section we have shown that System F is inadequate as a typed intermediate language for source languages that embody non-syntactic type equality — and Haskell has developed several such extensions. We have sketchily introduced System F_C as a solution to these difficulties. We will formalise it in the next section.

3. System $F_C(X)$

The main idea in $F_C(X)$ is that we pass around explicit evidence for type equalities, in just the same way that System F passes types explicitly. Indeed, in F_C the evidence γ for a type equality *is* a type; we use type abstraction for evidence abstraction, and type application for evidence application. Ultimately, we erase all types before running the program, and thereby erase all type-equality evidence as well, so the evidence passing has no run-time cost. However, that is not the only reason that it is better to represent evidence as a *type* rather than as a *term*, as we discuss in §3.10.

Figure 1 defines the syntax of System F_C, while Figures 2 and 3 give its static semantics. The notation $\overline{a}^n$ (where $n \geq 0$) means the sequence $a_1 \cdots a_n$; the "n" may be omitted when it is unimportant. Moreover, we use comma to mean sequence extension as follows: $\overline{a}^n, a_{n+1} \triangleq \overline{a}^{n+1}$. We use $fv(x)$ to denote the free variables of a structure x, which may be an expression, type term, or environment.

3.1 Conventional features

System F_C is a superset of System F. The syntax of types and kinds is given in Figure 1. Like F, F_C is impredicative, and has no stratification of types into polytypes and monotypes. The meta-variables φ, ρ, σ, τ, υ, and γ all range over types, and hence also over coercions. However, we adopt the convention that we use ρ, σ, τ, and υ in places where we can only have regular types (i.e., no coercions), and we use γ in places where we can only have coercion types. We use φ for types that can take either form. This choice of meta-variables is only a convention to aid the reader; formally, the coercion typing and kinding rules enforce the appropriate restrictions.

Our system allows types of higher kind; hence the type application form $\tau_1\ \tau_2$. However, like Haskell but unlike $F\omega$, our system has no type-level lambda, and type equality is syntactic identity (modulo alpha-conversion). This choice has pervasive consequences; it gives a remarkable combination of economy and expressiveness, but leaves some useful higher-kinded types out of reach. For example, there is no way to write the type constructor $(\lambda a.Either\ a\ Bool)$.

Value type constructors T range over (a) the built-in function type constructor, (b) any other built-in types, such as Int, and (c) algebraic data types. We regard a function type $\sigma_1 \to \sigma_2$ as the curried application of the built-in function type constructor to two arguments, thus $(\to)\ \sigma_1\ \sigma_2$. Furthermore, although we give the syntax of arrow types and quantified types in an uncurried way, we also sometimes use the following syntactic sugar:

$$\overline{\varphi}^n \to \varphi_r \quad\equiv\quad \varphi_1 \to \cdots \to \varphi_n \to \varphi_r$$
$$\forall \overline{\alpha}^n.\varphi \quad\equiv\quad \forall \alpha_1 \cdots \forall \alpha_n.\varphi$$

An *algebraic data type* T is introduced by a top-level $\textbf{data}$ declaration, which also introduces its *data constructors*. The type of a data constructor K takes the form

$$K : \forall \overline{a:\kappa.}^n \forall \overline{b:\iota}.\ \overline{\sigma} \to T\ \overline{a}^n$$

The first n quantified type variables $\overline{a}$ appear in the same order in the return type $T\ \overline{a}$. The remaining quantified type variables bind either existentially quantified type variables, or (as we shall see) coercions.

Types are classified by *kinds* κ, using the $\vdash_{\text{TY}}$ judgement in Figure 2. Temporarily ignoring the kind $\sigma_1 \sim \sigma_2$, the structure of kinds

Symbol Classes

$a, b, c, co \to \langle$type variable$\rangle$
$x, f \to \langle$term variable$\rangle$
$C \to \langle$coercion constant$\rangle$
$T \to \langle$value type constructor$\rangle$
$S_n \to \langle n$-ary type function$\rangle$
$K \to \langle$data constructor$\rangle$

Declarations

$pgm \to \overline{decl};\ e$
$decl \to \textbf{data}\ T : \overline{\kappa} \to \star\ \textbf{where}\ \overline{K : \forall \overline{a:\kappa}. \forall \overline{b:\iota}.\ \overline{\sigma} \to T\ \overline{a}}$
$\quad\mid\ \textbf{type}\ S_n : \overline{\kappa}^n \to \iota$
$\quad\mid\ \textbf{axiom}\ C : \sigma_1 \sim \sigma_2$

Sorts and kinds

δ	$\to$ TY $\mid$ CO	Sorts
κ, ι	$\to \star \mid \kappa_1 \to \kappa_2 \mid \sigma_1 \sim \sigma_2$	Kinds

Types and Coercions

d	$\to a \mid T$	Atom of sort TY
g	$\to c \mid C$	Atom of sort CO
$\varphi, \rho, \sigma, \tau, \upsilon, \gamma$	$\to a \mid C \mid T \mid \varphi_1\ \varphi_2 \mid S_n\ \overline{\varphi}^n \mid \forall a:\kappa.\ \varphi$	
	$\mid \text{sym}\,\gamma \mid \gamma_1 \circ \gamma_2 \mid \gamma @ \varphi \mid \text{left}\,\gamma \mid \text{right}\,\gamma$	
	$\mid \gamma \sim \gamma \mid \text{rightc}\,\gamma \mid \text{leftc}\,\gamma \mid \gamma \blacktriangleright \gamma$	

We use ρ, σ, τ, and υ for regular types, γ for coercions, and φ for both.

Syntactic sugar

Types $\kappa \Rightarrow \sigma \quad\equiv\quad \forall\_ : \kappa.\ \sigma$

Terms

u	$\to x \mid K$	Variables and data constructors
e	$\to u$	Term atoms
	$\mid \Lambda a:\kappa.\ e \mid e\ \varphi$	Type abstraction/application
	$\mid \lambda x:\sigma.\ e \mid e_1\ e_2$	Term abstraction/application
	$\mid \textbf{let}\ x:\sigma = e_1\ \textbf{in}\ e_2$	
	$\mid \textbf{case}\ e_1\ \textbf{of}\ \overline{p \to e_2}$	
	$\mid e \blacktriangleright \gamma$	Cast
p	$\to K\ \overline{b:\kappa}\ \overline{x:\sigma}$	Pattern

Environments

$\Gamma \to \epsilon \mid \Gamma, u:\sigma \mid \Gamma, d:\kappa \mid \Gamma, g:\kappa \mid \Gamma, S_n:\kappa$

A *top-level environment* binds only type constructors, T, S_n, data constructors K, and coercion constants C.

Figure 1: Syntax of System $F_C(X)$

$$\boxed{\Gamma \vdash_{\text{TY}} \sigma : \kappa}$$

$$\text{(TyVar)}\ \frac{d : \kappa \in \Gamma \quad \Gamma \vdash_k \kappa : \text{TY}}{\Gamma \vdash_{\text{TY}} d : \kappa} \qquad \text{(TyApp)}\ \frac{\Gamma \vdash_{\text{TY}} \sigma_1 : \kappa_1 \to \kappa_2 \quad \Gamma \vdash_{\text{TY}} \sigma_2 : \kappa_1}{\Gamma \vdash_{\text{TY}} \sigma_1\ \sigma_2 : \kappa_2}$$

$$\text{(TySCon)}\ \frac{(S_n : \overline{\kappa}^n \to \iota) \in \Gamma \quad \Gamma \vdash_{\text{TY}} \overline{\sigma : \kappa}^n}{\Gamma \vdash_{\text{TY}} S_n\ \overline{\sigma}^n : \iota} \qquad \text{(TyAll)}\ \frac{\Gamma, a : \kappa \vdash_{\text{TY}} \sigma : \star \quad \Gamma \vdash_k \kappa : \delta \quad a \notin fv(\Gamma)}{\Gamma \vdash_{\text{TY}} \forall a{:}\kappa.\, \sigma : \star}$$

$$\boxed{\Gamma \vdash_{\text{CO}} \gamma : \sigma \sim \tau}$$

$$\text{(CoRefl)}\ \frac{a{:}\kappa \in \Gamma \quad \Gamma \vdash_k \kappa : \text{TY}}{\Gamma \vdash_{\text{CO}} a : a \sim a} \qquad \text{(CoVar)}\ \frac{g{:}\sigma \sim \tau \in \Gamma}{\Gamma \vdash_{\text{CO}} g : \sigma \sim \tau}$$

$$\text{(CoAllT)}\ \frac{\Gamma, a{:}\kappa \vdash_{\text{CO}} \gamma : \sigma \sim \tau \qquad \Gamma \vdash_k \kappa : \text{TY} \quad a \notin fv(\Gamma)}{\Gamma \vdash_{\text{CO}} \forall a{:}\kappa.\, \gamma : \forall a{:}\kappa.\, \sigma \sim \forall a{:}\kappa.\, \tau} \qquad \text{(CoInstT)}\ \frac{\Gamma \vdash_{\text{CO}} \gamma : \forall a{:}\kappa.\, \sigma \sim \forall b{:}\kappa.\, \tau \qquad \Gamma \vdash_{\text{TY}} \upsilon : \kappa}{\Gamma \vdash_{\text{CO}} \gamma @ \upsilon : [\upsilon/a]\sigma \sim [\upsilon/b]\tau}$$

$$\text{(SComp)}\ \frac{\Gamma \vdash_{\text{CO}} \overline{\gamma : \sigma \sim \tau}^n \quad \Gamma \vdash_{\text{TY}} S_n\ \overline{\sigma}^n : \kappa}{\Gamma \vdash_{\text{CO}} S_n\ \overline{\gamma}^n : S_n\ \overline{\sigma}^n \sim S_n\ \overline{\tau}^n} \qquad \text{(Sym)}\ \frac{\Gamma \vdash_{\text{CO}} \gamma : \sigma \sim \tau}{\Gamma \vdash_{\text{CO}} \mathsf{sym}\, \gamma : \tau \sim \sigma} \qquad \text{(Trans)}\ \frac{\Gamma \vdash_{\text{CO}} \gamma_1 : \sigma_1 \sim \sigma_2 \qquad \Gamma \vdash_{\text{CO}} \gamma_2 : \sigma_2 \sim \sigma_3}{\Gamma \vdash_{\text{CO}} \gamma_1 \circ \gamma_2 : \sigma_1 \sim \sigma_3}$$

$$\text{(Comp)}\ \frac{\Gamma \vdash_{\text{CO}} \gamma_1 : \sigma_1 \sim \tau_1 \quad \Gamma \vdash_{\text{CO}} \gamma_2 : \sigma_2 \sim \tau_2 \qquad \Gamma \vdash_{\text{TY}} \sigma_1\ \sigma_2 : \kappa}{\Gamma \vdash_{\text{CO}} \gamma_1\ \gamma_2 : \sigma_1\ \sigma_2 \sim \tau_1\ \tau_2} \qquad \text{(Left)}\ \frac{\Gamma \vdash_{\text{CO}} \gamma : \sigma_1\ \sigma_2 \sim \tau_1\ \tau_2}{\Gamma \vdash_{\text{CO}} \mathsf{left}\, \gamma : \sigma_1 \sim \tau_1} \qquad \text{(Right)}\ \frac{\Gamma \vdash_{\text{CO}} \gamma : \sigma_1\ \sigma_2 \sim \tau_1\ \tau_2}{\Gamma \vdash_{\text{CO}} \mathsf{right}\, \gamma : \sigma_2 \sim \tau_2}$$

$$\text{(CompC)}\ \frac{\Gamma \vdash_{\text{CO}} \gamma : \kappa_1 \sim \kappa_2 \quad \Gamma \vdash_{\text{CO}} \gamma' : \sigma_1 \sim \sigma_2 \qquad \Gamma \vdash_k \kappa_1 : \text{CO}}{\Gamma \vdash_{\text{CO}} \gamma \Rightarrow \gamma' : (\kappa_1 \Rightarrow \sigma_1) \sim (\kappa_2 \Rightarrow \sigma_2)} \qquad \text{(LeftC)}\ \frac{\Gamma \vdash_{\text{CO}} \gamma : \begin{array}{c}\kappa_1 \Rightarrow \sigma_1 \\ \sim \\ \kappa_2 \Rightarrow \sigma_2\end{array}}{\Gamma \vdash_{\text{CO}} \mathsf{leftc}\, \gamma : \kappa_1 \sim \kappa_2} \qquad \text{(RightC)}\ \frac{\Gamma \vdash_{\text{CO}} \gamma : \begin{array}{c}\kappa_1 \Rightarrow \sigma_1 \\ \sim \\ \kappa_2 \Rightarrow \sigma_2\end{array}}{\Gamma \vdash_{\text{CO}} \mathsf{rightc}\, \gamma : \sigma_1 \sim \sigma_2}$$

$$(\sim)\ \frac{\Gamma \vdash_{\text{CO}} \gamma_1 : \sigma_1 \sim \tau_1 \quad \Gamma \vdash_{\text{CO}} \gamma_2 : \sigma_2 \sim \tau_2}{\Gamma \vdash_{\text{CO}} \gamma_1 \sim \gamma_2 : (\sigma_1 \sim \sigma_2) \sim (\tau_1 \sim \tau_2)} \qquad \text{(CastC)}\ \frac{\Gamma \vdash_{\text{CO}} \gamma_1 : \kappa \quad \Gamma \vdash_{\text{CO}} \gamma_2 : \kappa \sim \kappa'}{\Gamma \vdash_{\text{CO}} \gamma_1 \blacktriangleright \gamma_2 : \kappa'}$$

$$\boxed{\Gamma \vdash_e e : \sigma}$$

$$\text{(Var)}\ \frac{u : \sigma \in \Gamma}{\Gamma \vdash_e u : \sigma} \qquad \text{(Case)}\ \frac{\Gamma \vdash_e e : \sigma \quad \overline{\Gamma \vdash_p p \to e : \sigma \to \tau}}{\Gamma \vdash_e \mathbf{case}\ e\ \mathbf{of}\ \overline{p \to e} : \tau} \qquad \text{(Let)}\ \frac{\Gamma \vdash_e e_1 : \sigma_1 \quad \Gamma, x : \sigma_1 \vdash_e e_2 : \sigma_2}{\Gamma \vdash_e \mathbf{let}\ x{:}\sigma_1 = e_1\ \mathbf{in}\ e_2 : \sigma_2}$$

$$\text{(Cast)}\ \frac{\Gamma \vdash_e e : \sigma \quad \Gamma \vdash_{\text{CO}} \gamma : \sigma \sim \tau}{\Gamma \vdash_e e \blacktriangleright \gamma : \tau} \qquad \text{(Abs)}\ \frac{\Gamma \vdash_{\text{TY}} \sigma_x : \star \qquad \Gamma, x : \sigma_x \vdash_e e : \sigma}{\Gamma \vdash_e \lambda x{:}\sigma_x.\, e : \sigma_x \to \sigma} \qquad \text{(App)}\ \frac{\Gamma \vdash_e e_1 : \sigma_2 \to \sigma_1 \qquad \Gamma \vdash_e e_2 : \sigma_2}{\Gamma \vdash_e e_1\ e_2 : \sigma_1}$$

$$\text{(AbsT)}\ \frac{\Gamma, a : \kappa \vdash_e e : \sigma \quad \Gamma \vdash_k \kappa : \delta \quad a \notin fv(\Gamma)}{\Gamma \vdash_e \Lambda a{:}\kappa.\, e : \forall a{:}\kappa.\sigma} \qquad \text{(AppT)}\ \frac{\Gamma \vdash_e e : \forall a{:}\kappa.\sigma \quad \Gamma \vdash_k \kappa : \delta \quad \Gamma \vdash_\delta \varphi : \kappa}{\Gamma \vdash_e e\ \varphi : \sigma[\varphi/a]}$$

$$\boxed{\Gamma \vdash_p p \to e : \sigma \to \tau}$$

$$\text{(Alt)}\ \frac{K : \forall \overline{a{:}\kappa}.\forall \overline{b{:}\iota}.\overline{\sigma} \to T\ \overline{a} \in \Gamma \quad \theta = [\overline{\upsilon/a}] \quad \Gamma, \overline{b{:}\theta(\iota)}, \overline{x{:}\theta(\sigma)} \vdash_e e : \tau}{\Gamma \vdash_p K\ \overline{b{:}\theta(\iota)}\ \overline{x{:}\theta(\sigma)} \to e : T\ \overline{\upsilon} \to \tau}$$

$$\boxed{\Gamma \vdash decl : \Gamma'} \qquad \boxed{\Gamma \vdash pgm : \sigma}$$

$$\text{(Data)}\ \frac{\overline{\Gamma \vdash_{\text{TY}} \sigma : \star} \quad \Gamma \vdash_k \kappa : \text{TY}}{\Gamma \vdash (\mathbf{data}\ T{:}\kappa\ \mathbf{where}\ \overline{K{:}\sigma}) : (T{:}\kappa, \overline{K{:}\sigma})}$$

$$\text{(Type)}\ \frac{\Gamma \vdash_k \kappa : \text{TY}}{\Gamma \vdash (\mathbf{type}\ S : \kappa) : (S{:}\kappa)} \qquad \text{(Coerce)}\ \frac{\Gamma \vdash_k \kappa : \text{CO}}{\Gamma \vdash (\mathbf{axiom}\ C : \kappa) : (C{:}\kappa)}$$

$$\text{(Pgm)}\ \frac{\overline{\Gamma \vdash decl : \Gamma_d} \quad \Gamma = \Gamma_0, \overline{\Gamma_d} \qquad \Gamma \vdash e : \sigma}{\Gamma_0 \vdash \overline{decl};\ e : \sigma}$$

Figure 2: Typing rules for System $\mathrm{F_C}$(X)

$$\boxed{\Gamma \vdash_k \kappa : \delta}$$

$$\text{(Star)} \quad \frac{}{\Gamma \vdash_k \star : \text{TY}}$$

$$\text{(FunK)} \quad \frac{\Gamma \vdash_k \kappa_1 : \text{TY} \quad \Gamma \vdash_k \kappa_2 : \text{TY}}{\Gamma \vdash_k \kappa_1 \to \kappa_2 : \text{TY}}$$

$$\text{(EqTy)} \quad \frac{\Gamma \vdash_{\text{TY}} \sigma_1 : \kappa \quad \Gamma \vdash_{\text{TY}} \sigma_2 : \kappa}{\Gamma \vdash_k \sigma_1 \sim \sigma_2 : \text{CO}}$$

$$\text{(EqCo)} \quad \frac{\Gamma \vdash_k \gamma_1 : \text{CO} \quad \Gamma \vdash_k \gamma_2 : \text{CO}}{\Gamma \vdash_k \gamma_1 \sim \gamma_2 : \text{CO}}$$

Figure 3: Kinding rules for System $F_C(X)$

is conventional: $\star$ is the kind of proper types (that is, the types that a term can have), while higher kinds take the form $\kappa_1 \to \kappa_2$. Kinds guide type application by way of Rule (TyApp). Finally, the rules for judgements of the form $\Gamma \vdash_k \kappa : \delta$, given in Figure 3, ensure the well-formedness of kinds. Here δ is either TY for kinds formed from arrows and $\star$, or CO for coercion kinds of form $\sigma_1 \sim \sigma_2$. The conclusions of Rule (EqTy) and (EqCo) appear to overlap, but an actual implementation can deterministically decide which rule to apply, choosing (EqCo) iff γ_1 has the form $\varphi_1 \sim \varphi_2$.

The syntax of terms is largely conventional, as are their type rules which take the form $\Gamma \vdash_e e : \sigma$. As in F, every binder has an explicit type annotation, and type abstraction and application are also explicit. There is a **case** expression to take apart values built with data constructors. The patterns of a case expression are flat — there are no nested patterns — and bind existential type variables, coercion variables, and value variables. For example, suppose

$$K \;:\; \forall\, a{:}\star.\, \forall\, b{:}\star.\; a \;\to\; b \;\to\; (b \;\to\; Int) \to\; T \;\; a$$

Then a **case** expression that deconstructs K would have the form

$$\textbf{case}\; e \;\textbf{of}\; K\; (b{:}\star)\; (v{:}a)\; (x{:}b)\; (f{:}b \to Int) \to e'$$

Note that only the existential type variable b is bound in the pattern. To see why, one need only realise that K's type is isomorphic to:

$$K \;:\; \forall\, a{:}\star.\, (\exists\, b{:}\star.\, (a,\; b,\; (b \;\to\; Int))) \to\; T \;\; a$$

3.2 Type equality coercions

We now describe the unconventional features of our system. To begin with, consider the fragment of System F_C that omits type functions (i.e., **type** and **axiom** declarations). This fragment is sufficient to serve as a target for translating GADTs, and so is of interest in its own right. We return to type functions in §3.3.

The essential addition to plain F (beyond algebraic data types and higher kinds) is an infrastructure to construct, pass, and apply *type-equality coercions.* In F_C, a coercion, γ, is a special sort of type whose kind takes the unusual form $\sigma_1 \sim \sigma_2$. We can use such a coercion to cast an expression $e : \sigma_1$ to type σ_2 using the *cast expression* $(e \blacktriangleright \gamma)$; see Rule (Cast) in Figure 2. Our intuition for equality coercions is an *extensional* one:

> $\gamma : \sigma_1 \sim \sigma_2$ is evidence that a value of type σ_1 can be used in any context that expects a value of type σ_2, and vice versa.

By "can be used", we mean that running the program after type erasure will not go wrong. We stress that this is only an intuition; the soundness of our system is proved without appealing to any semantic notion of what $\sigma_1 \sim \sigma_2$ "means". We use the symbol "$\sim$" rather than "$=$", to avoid suggesting that the two types are intensionally equal.

Coercions are types – some would call them "constructors" [25, 12] since they certainly do not have kind $\star$ — and hence the term-level syntax for type abstraction and application ($\Lambda a.e$ and $e\ \varphi$) also serves for coercion abstraction and application. However, coercions have their own kinding judgement $\vdash_{\text{CO}}$, given in Figure 2. The type of a term often has the form $\forall co{:}(\sigma_1 \sim \sigma_2).\varphi$, where φ does not mention co. We allow the standard syntactic sugar for this case, writing it thus: $(\sigma_1 \sim \sigma_2) \Rightarrow \varphi$ (see Figure 1). Incidentally, note that although coercions are types, they do not classify values. This is standard in F_ω; for example, there are no values whose type has kind $\star \to \star$.

More complex coercions can be built by combining or transforming other coercions, such that every syntactic form corresponds to an inference rule of equational logic. We have the reflexivity of equality for a given type σ (witnessed by the type itself), symmetry '$\text{sym}\,\gamma$', transitivity '$\gamma_1 \circ \gamma_2$', type composition '$\gamma_1\ \gamma_2$', and decomposition '$\text{left}\,\gamma$' and '$\text{right}\,\gamma$'. The typing rules for these coercion expressions are given in Figure 2.

Here is an example, taken from §2. Suppose a GADT $Expr$ has a constructor $Succ$ with type

$$Succ \;:\; \forall\; a{:}\star.\, (a \sim Int) \;\Rightarrow\; Exp\ Int \;\to\; Exp\ a$$

(notice the use of the syntactic sugar $\kappa \Rightarrow \sigma$). Then we can construct a value of type $Exp\ Int$ thus: $Succ\ Int\ Int\ e$. The second argument Int is a regular type used as a coercion witnessing reflexivity — i.e., we have $Int : (Int \sim Int)$ by Rule (CoRefl). Rule (CoRefl) itself only covers type variables and constructors, but in combination with Rule (Comp), the reflexivity of complex types is also covered. More interestingly, here is a function that decomposes a value of type $Exp\ a$:

$$\begin{array}{l} foo \;:\; \forall\; a{:}\star.\; Exp\ a \;\to\; a \;\to\; a \\ \;= \Lambda a{:}\star.\, \lambda e{:}Exp\ a.\, \lambda x{:}a. \\ \quad \textbf{case}\; e\; \textbf{of} \\ \qquad Succ\ (co{:}a \sim Int)\ (e'{:}Exp\ Int) \;\to \\ \qquad\qquad (foo\ Int\ e'\ 0 \;+\; (x \blacktriangleright co)) \blacktriangleright \text{sym}\ co \end{array}$$

The **case** pattern binds the coercion co, which provides evidence that a and Int are the same type. This evidence is needed twice, once to cast $x : a$ to Int, and once to coerce the Int result back to a, via the coercion $(\text{sym}\, co)$.

Coercion composition allows us to "lift" coercions through arbitrary types, in the style of logical relations [1]. For example, if we have a coercion $\gamma{:}(\sigma_1 \sim \sigma_2)$ then the coercion $Tree\ \gamma$ is evidence that $Tree\ \sigma_1 \sim Tree\ \sigma_2$, using rules (Comp) and (CoRefl) and (CoVar). More generally, our system has the following theorem.

THEOREM 1 (Lifting). *If $\Gamma' \vdash_{\text{CO}} \gamma : \sigma_1 \sim \sigma_2$ and $\Gamma \vdash_{\text{TY}} \varphi : \kappa$, then $\Gamma' \vdash_{\text{CO}} [\gamma/a]\varphi : [\sigma_1/a]\varphi \sim [\sigma_2/a]\varphi$, for any type φ, including polytypes, where $\Gamma = \Gamma', a{:}\kappa'$ such that a does not appear in Γ'.*

PROOF. The first task is to show that $\Gamma \vdash_{\text{CO}} \varphi : \varphi \sim \varphi$ (1) for all (well-formed) types φ (proof by induction on φ). Then, we can derive $\Gamma \vdash_{\text{CO}} [\gamma/a]\varphi : [\sigma_1/a]\varphi \sim [\sigma_2/a]\varphi$ from the derivation for (1) by replacing each (CoRefl) step with the derivation steps for $\Gamma \vdash_{\text{CO}} \gamma : \sigma_1 \sim \sigma_2$. $\square$

For example, if $\gamma : \sigma_1 \sim \sigma_2$ then

$$\forall\, b.\, \gamma \;\to\; Int \;\;:\; (\forall\, b.\, \sigma_1 \;\to\; Int) \sim (\forall\, b.\, \sigma_2 \;\to\; Int)$$

Dually decomposition enables us to take evidence apart. For example, assume $\gamma{:}\, Tree\ \sigma_1 \sim Tree\ \sigma_2$; then, $(\text{right}\,\gamma)$ is evidence that $\sigma_1 \sim \sigma_2$, by rule (Right). Decomposition is necessary for the translation of GADT programs to F_C, but is problematic in earlier approaches [3, 9]. The soundness of decomposition relies, of course, on algebraic types being injective; i.e., $\mathit{Tree}\ \sigma_1 = \mathit{Tree}\ \sigma_2$ iff $\sigma_1 = \sigma_2$. Notice, too, that Tree by itself is a coercion relating two types of higher kind.

Similarly, one can compose and decompose equalities over polytypes, using rules (CoAllT) and (CoInstT). For example,

$$\begin{array}{l}\gamma : (\forall a.a \rightarrow \mathit{Int}) \sim (\forall a.a \rightarrow b) \\ \quad \vdash_{\mathrm{co}} \gamma@\mathit{Bool}\ :\ (\mathit{Bool} \rightarrow \mathit{Int}) \sim (\mathit{Bool} \rightarrow b)\end{array}$$

This simply says that if the two polymorphic functions are interchangeable, then so are their instantiations at Bool.

Rules (CompC), (LeftC), and (RightC) are analogous to (Comp), (Left), and (Right): they allow composition and decomposition of a type of form $\kappa \Rightarrow \varphi$, where κ is a coercion kind. These rules are essential to allow us to validate this consequence of Theorem 1:

$$\gamma : \sigma_1 \sim \sigma_2 \vdash_{\mathrm{co}} (\gamma \sim \mathit{Int} \Rightarrow \mathit{Tree}\ \gamma)\ :\ \begin{array}{c}\sigma_1 \sim \mathit{Int} \Rightarrow \mathit{Tree}\ \sigma_1 \\ \sim \\ \sigma_2 \sim \mathit{Int} \Rightarrow \mathit{Tree}\ \sigma_2\end{array}$$

Even though $\kappa \Rightarrow \varphi$ is is sugar for $\forall\_ : \kappa.\varphi$, we cannot generalise (CoAllT) to cover (CompC) because the former insists that the two kinds are identical.

We will motivate the need for rules ($\sim$) and (CastC) when discussing the dynamic semantics (§3.7).

3.3 Type functions

Our last step extends the power of $\mathrm{F_C}$ by adding *type functions* and *equality axioms*, which are crucial for translating associated types, functional dependencies, and the like. A type function S_n is introduced by a top-level **type** declaration, which specifies its *kind* $\overline{\kappa}^n \rightarrow \iota$, but says nothing about its *interpretation*. The index n indicates the *arity* of S. The syntax of types requires that S_n always appears applied to its full complement of n arguments (§3.6 explains why). The arity subscript should be considered part of the name of the type constructor, although we will often elide it, writing $\mathit{Elem}\ \sigma$ rather than $\mathit{Elem}_1\ \sigma$, for example.

A type function is given its interpretation by one or more equality **axiom**s. Each axiom introduces a coercion constant, whose kind specifies the types between which the coercion converts. Thus:

$$\mathbf{axiom}\ \mathit{elemBitSet}\ :\ \mathit{Elem}\ \mathit{BitSet} \sim \mathit{Char}$$

introduces the named coercion constant $\mathit{elemBitSet}$. Given an expression $e : \mathit{Elem}\ \mathit{BitSet}$, we can use the axiom via the coercion constant as in the cast $e \blacktriangleright \mathit{elemBitSet}$, which is of type Char.

We often want to state axioms involving parametric types, thus:

$$\mathbf{axiom}\ \mathit{elemList}\ :\ (\forall e{:}\star.\ \mathit{Elem}\ [e]) \sim (\forall e{:}\star.\ e)$$

This is the axiom generated from the instance declaration for $\mathit{Collects}\ [e]$ in §2.2. To use this axiom as a coercion, say, for lists of integers, we need to apply the coercion constant to a type argument:

$$\mathit{elemList}\ \mathit{Int}\ :\ (\mathit{Elem}\ [\mathit{Int}] \sim \mathit{Int})$$

which appeals to Rule (CoInstT) of Figure 2. We have already seen the usefulness of (CoInstT) towards the end of §3.2, and here we simply re-use it. It may be surprising that we use one quantifier on each side of the equality, instead of quantifying over the entire equality as in

$$\forall a{:}\star.\ (\mathit{Elem}\ [a] \sim a) \qquad \text{-- Not well-formed } \mathrm{F_C}!$$

One could add such a construct, but it is simply unnecessary. We already have enough machinery, and when thought of as a logical relation, the form with a quantifier on each side makes perfect sense.

3.4 Type functions are open

A crucial property of type functions is that they are *open*, or *extensible*. A type function S may take an argument of kind $\star$ (or $\star \rightarrow \star$, etc), and, since the kind $\star$ is extensible, we cannot write out all the cases for S at the moment we introduce S. For example, imagine that a library module contains the definition of the Collects class (§2.2). Then a client imports this module, defines a new type T (thereby adding a new constant to the extensible kind $\star$), and wants to make T an instance of Collects. In $\mathrm{F_C}$ this is easy by simply writing in the client module

```
import CollectsLib
instance Collects T where {type Elem T = E; ...}
```

where we assume that E is the element type of the collection type T. In short, open type functions are absolutely required to support modular extensibility.

We do not argue that *all* type functions should be open; it would certainly be possible to extend $\mathrm{F_C}$ with non-extensible kind declarations and closed type functions. Such an extension would be useful; consider the well-worn example of lists parametrised by their length, which we give in source-code form to reduce clutter:

```
kind Nat = Z | S Nat

data Seq a (n::Nat) where
  Nil  :: Seq a Z
  Cons :: a -> Seq a n -> Seq a (S n)

app :: Seq a n -> Seq a m -> Seq a (Plus n m)
app Nil         ys = ys
app (Cons x xs) ys = Cons x (app xs ys)

type Plus :: Nat -> Nat -> Nat
     Plus Z     b = b
     Plus (S a) b = S (Plus a b)
```

Whilst we can translate this into $\mathrm{F_C}$, we would be forced to give Plus the kind $\star \rightarrow \star \rightarrow \star$, which allows nonsensical forms like $\mathit{Plus}\ \mathit{Int}\ \mathit{Bool}$. Furthermore, the non-extensibility of Nat would allow induction, which is not available in $\mathrm{F_C}$ precisely because kind $\star$ is extensible.

Other closely-related languages support closed type functions; for example LH [25], LX [12], and Ωmega [37]. In this paper, however, we focus on open-ness, since it is one of $\mathrm{F_C}$'s most distinctive features and is crucial to translating associated types.

3.5 Consistency

In System $\mathrm{F_C}$(X), we refine the equational theory of types by giving non-standard equality axioms. *So what is to prevent us declaring unsound axioms?* For example, one could easily write a program that would crash, using the coercion constant introduced by the following axiom:

$$\mathbf{axiom}\ \mathit{utterlybogus}\ :\ \mathit{Int} \sim \mathit{Bool}$$

(where Int and Bool are both algebraic data types). There are many *ad hoc* ways to restrict the system to exclude such cases. The most general way is this: we require that the axioms, taken together, are *consistent*. We essentially adapt the standard notion of consistency of sets of equations [13, Section 3] to our setting.

DEFINITION 1 (Value type). *A type σ is a* value type *if it is of form $\forall a.\upsilon$ or $T\ \overline{\upsilon}$.*

DEFINITION 2 (Consistency). *Γ is* consistent *iff*

1. *If $\Gamma \vdash_{\mathrm{co}} \gamma : T\ \overline{\sigma} \sim \upsilon$, and υ is a value type, then $\upsilon = T\ \overline{\tau}$.*
2. *If $\Gamma \vdash_{\mathrm{co}} \gamma : \forall a{:}\kappa.\sigma \sim \upsilon$, and υ is a value type, then $\upsilon = \forall a{:}\kappa.\tau$.*

That is, if there is a coercion connecting two *value* types — algebraic data types, built-in types, functions, or foralls — then the outermost type constructors must be the same. For example, there can be no coercion of type $Bool \sim Int$. It is clear that the consistency of Γ is necessary for soundness, and it turns out that it is also sufficient (§3.7).

Consistency is only required of the *top-level* environment, however (Figure 1). For example, consider this function:

$$f = \lambda(g\!:\!Int \sim Bool).\, 1 + (True \blacktriangleright g)$$

It uses the bogus coercion g to cast an Int to a $Bool$, so f would crash if called. But there is no problem, because *the function can never be called*; to do so, one would have to produce evidence that Int and $Bool$ are interchangeable. The proof in §3.7 substantiates this intuition.

Nevertheless, consistency is absolutely required for the top-level environment, but alas it is an undecidable property. That is why we call the system "$\mathrm{F_C}$(X)": it is parametrised by a decision procedure X for determining consistency. There is no "best" choice for X, so instead of baking a particular choice into the language, we have left the choice open. Each particular source-program construct that exploits type equalities comes with its own decision procedure — or, alternatively, guarantees *by construction* to generate only consistent axioms, so that consistency need never be checked. All the applications we have implemented so far take the latter approach. For example, GADTs generate no axioms at all (Section 4); newtypes generate exactly one axiom per newtype; and associated types are constrained to generate a non-overlapping rewrite system (Section 5).

3.6 Saturation of type functions

We remarked earlier that applications of type functions S_n are required to be saturated. The reason for this insistence is, again, consistency. We definitely want to allow abstract types to be non-injective; for example:

$$\begin{array}{l} \mathbf{axiom}\ c1\ :\ S_1\ Int \sim Bool \\ \mathbf{axiom}\ c2\ :\ S_1\ Bool \sim Bool \end{array}$$

Here, both $S_1\ Int$ and $S_1\ Bool$ are represented by the $Bool$ type. But now we can form the coercion $(c1 \circ (\mathrm{sym}\, c2))$ which has type $S_1\ Int \sim S_1\ Bool$, and from that we must not be able to deduce (via right) that $Int \sim Bool$, because that would violate consistency! Applications of type functions are therefore syntactically distinguished so that right and left apply only to ordinary type application (Rules (Left) and (Right) in Figure 2), and not to applications of type functions. The same syntactic mechanism prevents a partial type-function application from appearing as a type argument, thereby instantiating a type variable with a partial application — in effect, type variables of higher-kind range only over injective type constructors.

However, it is perfectly acceptable for a type function to have an arity of 1, say, but a higher kind of $\star \to \star \to \star$. For example:

$$\begin{array}{l} \mathbf{type}\ HS_1\ :\ \star\ \to\ \star\ \to\ \star \\ \mathbf{axiom}\ c1\ :\ HS_1\ Int \sim [\,] \\ \mathbf{axiom}\ c2\ :\ HS_1\ Bool \sim Maybe \end{array}$$

An application of HS to one type is saturated, although it has kind $\star \to \star$ and can be applied (via ordinary type application) to another type.

3.7 Dynamic semantics and soundness

The operational semantics of $\mathrm{F_C}$ is shown in Figure 4. In the expression reductions we omit the type annotations on binders to save clutter, but that is mere abbreviation.

An unusual feature of our system, which we share with Crary's coercion calculus for inclusive subtyping [11], is that values are stratified into *cvalues* and *plain values*; their syntax is in Figure 4. Evaluation reduces a closed term to a *cvalue*, or diverges. A cvalue is either a *plain value* v (an abstraction or saturated constructor application), or it is a value wrapped in a single cast, thus $v \blacktriangleright \gamma$ (Figure 4). The latter form is needed because we cannot reduce a term to a plain value without losing type preservation; for example, we cannot reduce $(True \blacktriangleright \gamma)$, where $\gamma\!:\!Bool \sim S$ any further without changing its type from S to $Bool$.

However, there are four situations when a cvalue will not do, namely as the function part of a type, coercion, or function application, or as the scrutinee of a **case** expression. Rules (TPush), (CPush), (Push) and (KPush) deal with those situations, by pushing the coercion inside the term, turning the cast into a plain value. Notice that all four rules leave the *context* (the application or case expression) unchanged; they rewrite the function or case scrutinee respectively. Nevertheless, the context is necessary to guarantee that the type of the rewritten term is a function or data type respectively.

Rules (TPush) and (Push) are quite straightforward. Rule (CPush) is rather like (Push), but at the level of coercions. It is this rule that forces us to add the forms $(\gamma_1 \sim \gamma_2)$, $(\gamma_1 \blacktriangleright \gamma_2)$, $(\mathrm{leftc}\,\gamma)$, and $(\mathrm{rightc}\,\gamma)$ to the language of coercions. We will shortly provide an example to illustrate this point.

The final rule, (KPush), is more complicated. Here is an example, stripped of the **case** context, where $Cons : \forall a.a \to [a] \to [a]$, and $\gamma : [Int] \sim [S\ Bool]$:

$$\begin{array}{l} (Cons\ Int\ e_1\ e_2)\blacktriangleright\gamma \longrightarrow Cons\ (S\ Bool)\ \ (e_1 \blacktriangleright \mathrm{right}\,\gamma) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (e_2 \blacktriangleright ([\,])\ (\mathrm{right}\,\gamma)) \end{array}$$

The coercion wrapped around the application of $Cons$ is pushed inside to wrap each of its components. (Of course, an implementation does none of this, because types and coercions are erased.) The type preservation of this rule relies on Theorem 1 in Section 3.2, which guarantees that $e_i \blacktriangleright \theta(\rho_i)$ has the correct type.

The rule is careful to cast the *coercion* arguments as well as the *value* arguments. Here is an example, taken from Section 2.3:

$$\begin{array}{rcl} F & : & \forall a\ b.(b \sim F1\ a) \Rightarrow FDict\ a\ b \\ \gamma & : & FDict\ Int\ Bool \sim FDict\ c\ d \\ \varphi & : & Bool \sim F1\ Int \end{array}$$

Now, stripped of the **case** context, rule (KPush) describes the following transition:

$$(F\ Int\ Bool\ \varphi) \blacktriangleright \gamma \ \ \longrightarrow\ \ F\ c\ d\ (\varphi \blacktriangleright (\gamma_2 \sim F1\ \ \gamma_1))$$

where $\gamma_1 = \mathrm{right}\,(\mathrm{left}\,\gamma)$ and $\gamma_2 = \mathrm{right}\,\gamma$. The coercion argument φ is cast by the strange-looking coercion $\gamma_2 \sim F1\,\gamma_1$, whose kind is $(Bool \sim F1\ Int) \sim (d \sim F1\ c)$. That is why we need rule ($\sim$) in Figure 2, so that we can type such coercions.

We derived all three "push" rules in a systematic way. For example, for (Push) we asked what e' (involving e and γ) would ensure that $((\lambda x.e) \blacktriangleright \gamma) = \lambda y.e'$. The reader may like to check that if the left-hand side of each rule is well-typed (in the top-level context) then so is the right-hand side.

When a data constructor has a higher-rank type, in which the argument types are themselves quantified, a good deal of book-keeping is needed. For example, suppose that

$$\begin{array}{rcl} K & : & \forall a\!:\!*.\,(a \sim Int \Rightarrow a \to Int) \to T\ a \\ \gamma & : & T\ \sigma_1 \sim T\ \sigma_2 \\ e & : & (\sigma_1 \sim Int) \Rightarrow \sigma_1 \to Int \end{array}$$

Then, according to rule (KPush) we find (as before we strip off the **case** context)

$$(K\ \sigma_1\ e) \blacktriangleright \gamma \quad \longrightarrow \quad K\ \sigma_2\ (e \blacktriangleright \gamma')$$

where $\gamma' = (\mathrm{right}\,\gamma \sim Int) \Rightarrow \mathrm{right}\,\gamma \to Int$, which is obtained by substituting $[\mathrm{right}\,\gamma/a]$ in $(a \sim Int) \Rightarrow a \to Int$.

Values:

$$\begin{array}{llll} \text{Plain values} & v & ::= & \Lambda a.e \mid \lambda x.e \mid K\ \overline{\sigma}\ \overline{\varphi}\ \overline{e} \\ \text{Cvalues} & cv & ::= & v \blacktriangleright \gamma \mid v \end{array}$$

Evaluation contexts:

$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']} \qquad E \quad ::= \quad [\,] \mid E\ e \mid E\ \tau \mid E \blacktriangleright \gamma \mid \textbf{case}\ E\ \textbf{of}\ \overline{p \to rhs}$$

Expression reductions:

$$\begin{array}{llll}
\text{(TBeta)} & (\Lambda a.e)\ \varphi & \longrightarrow & [\varphi/a]e \\
\text{(Beta)} & (\lambda x.e)\ e' & \longrightarrow & [e'/x]e \\
\text{(Case)} & \textbf{case}\ (K\ \overline{\sigma}\ \overline{\varphi}\ \overline{e})\ \textbf{of}\ \ldots\ K\ \overline{b}\ \overline{x} \to e'\ \ldots & \longrightarrow & [\overline{\varphi/b}, \overline{e/x}]e' \\
\text{(Comb)} & (v \blacktriangleright \gamma_1) \blacktriangleright \gamma_2 & \longrightarrow & v \blacktriangleright (\gamma_1 \circ \gamma_2)
\end{array}$$

$$\begin{array}{llll}
\text{(TPush)} & ((\Lambda a\!:\!\kappa.\,e) \blacktriangleright \gamma)\ \varphi & \longrightarrow & (\Lambda a\!:\!\kappa.\,(e \blacktriangleright \gamma@a))\ \varphi \\
& & \text{where} & \gamma : (\forall a\!:\!\kappa.\,\sigma_1) \sim (\forall b\!:\!\kappa.\,\sigma_2)
\end{array}$$

$$\begin{array}{lllll}
\text{(CPush)} & ((\Lambda a\!:\!\kappa.\,e) \blacktriangleright \gamma)\ \varphi & \longrightarrow & (\Lambda a'\!:\!\kappa'.\,(([(a' \blacktriangleright \gamma_1)/a]e) \blacktriangleright \gamma_2))\ \varphi & \\
& & \text{where} & \gamma : (\kappa \Rightarrow \sigma) \sim (\kappa' \Rightarrow \sigma') & \\
& & & \gamma_1 = \text{sym}\,(\text{leftc}\,\gamma) & \text{– coercion for argument} \\
& & & \gamma_2 = \text{rightc}\,\gamma & \text{– coercion for result}
\end{array}$$

$$\begin{array}{lllll}
\text{(Push)} & ((\lambda x.e) \blacktriangleright \gamma)\ e' & \longrightarrow & (\lambda y.([(y \blacktriangleright \gamma_1)/x]e \blacktriangleright \gamma_2))\ e' & \\
& & \text{where} & \gamma_1 = \text{sym}\,(\text{right}\,(\text{left}\,\gamma)) & \text{– coercion for argument} \\
& & & \gamma_2 = \text{right}\,\gamma & \text{– coercion for result}
\end{array}$$

$$\begin{array}{llll}
\text{(KPush)} & \textbf{case}\ (K\ \overline{\sigma}\ \overline{\varphi}\ \overline{e} \blacktriangleright \gamma)\ \textbf{of}\ \overline{p \to rhs} & \longrightarrow & \textbf{case}\ (K\ \overline{\tau}\ \overline{\varphi'}\ \overline{e'})\ \textbf{of}\ \overline{p \to rhs} \\
& & \text{where} & \gamma : T\ \overline{\sigma} \sim T\ \overline{\tau} \\
& & & K : \forall \overline{a\!:\!\kappa}.\forall \overline{b\!:\!\iota}.\ \overline{\rho} \to T\ \overline{a}^n \\
& & & \varphi_i' = \begin{cases} \varphi_i \blacktriangleright \theta(\upsilon_1 \sim \upsilon_2) & \text{if } b_i : \upsilon_1 \sim \upsilon_2 \\ \varphi_i & \text{otherwise} \end{cases} \\
& & & e_i' = e_i \blacktriangleright \theta(\rho_i) \\
& & & \theta = [\overline{\gamma_i/a_i}, \overline{\varphi_i/b_i}] \\
& & & \gamma_i = \text{right}\,\underbrace{(\text{left} \ldots (\text{left}}_{n-i}\ \gamma))
\end{array}$$

Figure 4: Operational semantics

Now suppose that we later reduce the (sub)-expression

$$(e \blacktriangleright \gamma')\ \gamma''$$

where $e = \Lambda\, b\!:\!(\sigma_1 \sim \mathit{Int}).\ \lambda\, x\!:\!\sigma_1.\ x \blacktriangleright b$. Before we can apply rule (CPush) we have to determine the kind of γ'. It is straightforward to deduce that

$$\gamma' : (\sigma_1 \sim \mathit{Int} \Rightarrow \sigma_1 \to \mathit{Int}) \sim (\sigma_2 \sim \mathit{Int} \Rightarrow \sigma_2 \to \mathit{Int})$$

Hence, via (CPush) we find that

$$\begin{array}{ll} & ((\Lambda b\!:\!(\sigma_1 \sim \mathit{Int}).\ \lambda x\!:\!\sigma_1.\ x \blacktriangleright b) \blacktriangleright \gamma')\ \gamma'' \\ \longrightarrow & (\Lambda c\!:\!(\sigma_2 \sim \mathit{Int}).\ (\lambda x\!:\!\sigma_1.\ x \blacktriangleright (c \blacktriangleright \gamma_1)) \blacktriangleright \gamma_2)\ \gamma'' \end{array}$$

where $\gamma_1 = \text{sym}\,(\text{leftc}\,\gamma')$, $\gamma_1 : (\sigma_1 \sim \mathit{Int}) \sim (\sigma_2 \sim \mathit{Int})$, $\gamma_2 = \text{rightc}\,\gamma'$ and $\gamma_2 : (\sigma_1 \to \mathit{Int}) \sim (\sigma_2 \to \mathit{Int})$.

Notice that forms $(\gamma_1 \sim \gamma_2)$, $(\gamma_1 \blacktriangleright \gamma_2)$, $(\text{leftc}\,\gamma)$, and $(\text{rightc}\,\gamma)$ only appear during the reduction of $\mathrm{F_C}$ programs. In case we restrict $\mathrm{F_C}$ types to be rank 1 none of these forms are necessary.

THEOREM 2 (Progress and subject reduction). *Suppose that a top-level environment Γ is consistent, and $\Gamma \vdash_e e : \sigma$. Then either e is a cvalue, or $e \longrightarrow e'$ and $\Gamma \vdash_e e' : \sigma$ for some term e'.*

PROOF. By structural induction on e. The interesting case is for application. Suppose $\Gamma \vdash_e e_1\ e_2 : \sigma$. Then $\Gamma \vdash_e e_1 : \tau \to \sigma$ and $\Gamma \vdash_e e_2 : \tau$. Then there are three well-typed possibilities for e:

1. e_1 is not a cvalue. Then by the induction hypothesis, e_1 can take a (type-preserving) step.
2. e_1 is is a plain value which, to be well typed, must be of form $\lambda x.e_3$. Hence we can take a (Beta) step.
3. e_1 is $v \blacktriangleright \gamma$. By consistency v must have a function type. Since v is a value, v must be of form $\lambda x.e_3$, so we can take a type-preserving step using (Push).

The other cases can be proved in a similar way. For example, suppose $\Gamma \vdash_e \textbf{case}\ e\ \textbf{of}\ \overline{p \to e} : \tau$. Then $\Gamma \vdash_e e : \sigma$ and $\Gamma \vdash_p p \to e : \sigma \to \tau$. As before, we can distinguish among the following three well-typed possibilities for case expressions:

1. e is not a cvalue. Then by the induction hypothesis, e can take a (type-preserving) step.
2. e is a plain value which, to be well typed, must be of form $K\ \overline{\sigma'}\ \overline{\varphi}\ \overline{e'}$. Hence we can take a (Case) step (we assume that case expressions have exhaustive alternatives).
3. e is $v \blacktriangleright \gamma$ where $\gamma : T\ \overline{\sigma'} \sim T\ \overline{\tau'}$ (i.e.$\sigma = T\ \overline{\tau'}$). By consistency and since v is a value, v must be of form $K\ \overline{\sigma'}\ \overline{\varphi}\ \overline{e'}$ where $K : \forall \overline{a\!:\!\kappa}.\forall \overline{b\!:\!\iota}.\ \overline{\rho} \to T\ \overline{a}$. It is straightforward to verify that $K\ \overline{\tau'}\ \overline{\varphi'}\ \overline{e''}$ is of type $T\ \overline{\tau'}$ where

$$\begin{array}{l}
\varphi_i' = \begin{cases} \varphi_i \blacktriangleright \theta(\upsilon_1 \sim \upsilon_2) & \text{if } b_i : \upsilon_1 \sim \upsilon_2 \\ \varphi_i & \text{otherwise} \end{cases} \\
e_i'' = e_i' \blacktriangleright \theta(\rho_i) \\
\theta = [\overline{\gamma_i/a_i}, \overline{\varphi_i/b_i}] \\
\gamma_i = \text{right}\,\underbrace{(\text{left} \ldots (\text{left}}_{n-i}\ \gamma))
\end{array}$$

Hence, we can take a type-preserving step using (KPush).

□

COROLLARY 1 (Syntactic Soundness). *Let* Γ *be consistent top-level environment and* $\Gamma \vdash_e e : \sigma$. *Then either* $e \longrightarrow^* cv$ *and* $\Gamma \vdash_e cv : \sigma$ *for some cvalue* cv*, or the evaluation diverges.*

We give a call-by-name semantics here, but a call-by-value semantics would be equally easy: simply extend the syntax of evaluation contexts with the form $v\ E$, and restrict the argument of rule (Beta) to be a cvalue.

In general, evaluation affects *expressions* only, not types. Since coercions are types, it follows that coercions are not evaluated either. This means that we can completely avoid the issue of normalisation of coercions, what a coercion "value" might be, and so on.

3.8 Robustness to transformation

One of our major concerns was to make it easy for a compiler to transform and optimise the program. For example, consider this fragment:

$$\lambda x.\, \textbf{case}\ x\ \textbf{of}\ \{\ T1\ \rightarrow\ \textbf{let}\ z\ =\ y+1\ \textbf{in}\ ...;\ ...\ \}$$

A good optimisation might be to float the let-binding out of the lambda, thus:

$$\textbf{let}\ z\ =\ y+1\ \textbf{in}\ \lambda x.\, \textbf{case}\ x\ \textbf{of}\ \{\ T1\ \rightarrow\ ...;\ ...\ \}$$

But suppose that $x : T a$ and $y : a$, and that the pattern match on $T1$ refines a to Int. Then the floated form is type-incorrect, because the let-binding is now out of the scope of the pattern match. This is a real problem for any intermediate language in which equality is implicit. In $\mathrm{F_C}$, however, y will be cast to Int using a coercion that is bound by the pattern match on $T1$. So the type-incorrect transformation is prevented, because the binding mentions a variable bound by the match; and better still, we can perform the optimisation in a type-correct way by abstracting over the variable to get this:

$$\begin{array}{l}\textbf{let}\ z'\ =\ \lambda g.\, (y \blacktriangleright g)\ +\ 1\\ \textbf{in}\ \lambda x.\, \textbf{case}\ x\ \textbf{of}\ \{\ T1\ g\ \rightarrow\ \textbf{let}\ z\ =\ z'\ g\ \textbf{in}\ ...;\ ...\ \}\end{array}$$

The inner let-binding obviously cannot be floated outside, because it mentions a variable bound by the match.

Another useful transformation is this:

$$(\textbf{case}\ x\ \textbf{of}\ p_i\ \rightarrow\ e_i)\ arg\ \ =\ \ \textbf{case}\ x\ \textbf{of}\ p_i\ \rightarrow\ e_i\ arg$$

This is valid in $\mathrm{F_C}$, but not in the more implicit language LH, for example [25].

In summary, we believe that $\mathrm{F_C}$'s obsessively-explicit form makes it easy to write type-preserving transformations, whereas doing so is significantly harder in a language where type equality is more implicit.

3.9 Type and coercion erasure

System $\mathrm{F_C}$ permits syntactic type erasure much as plain System F does, thereby providing a solid guarantee that coercions impose absolutely no run-time penalty. Like types, coercions simply provide a statically-checkable guarantee that there will be no run-time crash.

Formally, we can define an erasure function e°, which erases all types and coercions from the term, and prove the standard erasure theorem. Following Pierce [32, Section 23.7] we erase a type abstraction to a trivial term abstraction, and type application to term application to the unit value; this standard device preserves termination behaviour in the presence of `seq`, or with call-by-value semantics. The only difference from plain F is that we also erase casts.

$$\begin{array}{rcl@{\qquad}rcl} x^\circ &=& x & (\lambda x\!:\!\varphi.\, e)^\circ &=& \lambda x.e^\circ\\ K^\circ &=& K & (e_1\ e_2)^\circ &=& {e_1}^\circ {e_2}^\circ\\ (\Lambda a\!:\!\kappa.\, e)^\circ &=& \lambda a.e^\circ & (e \blacktriangleright \gamma)^\circ &=& e^\circ\\ (e\ \sigma)^\circ &=& e^\circ() & (K\ \overline{a\!:\!\kappa}\ \overline{x\!:\!\varphi})^\circ &=& K\ \overline{a}\ \overline{x} \end{array}$$

$$\begin{array}{rcl} (\textbf{let}\ x\!:\!\sigma = e_1\ \textbf{in}\ e_2)^\circ &=& \textbf{let}\ x = {e_1}^\circ\ \textbf{in}\ {e_2}^\circ\\ (\textbf{case}\ e_1\ \textbf{of}\ \overline{p \rightarrow e_2})^\circ &=& \textbf{case}\ {e_1}^\circ\ \textbf{of}\ \overline{p^\circ \rightarrow {e_2}^\circ} \end{array}$$

THEOREM 3. *Suppose that a top-level environment* Γ *is consistent, and* $\Gamma \vdash_e e_1 : \sigma$. *Then, (a) either* e_1 *is a cvalue and* ${e_1}^\circ$ *is a value or (b) we have* $e_1 \longrightarrow e_2$ *and either* ${e_1}^\circ \longrightarrow {e_2}^\circ$ *or* ${e_1}^\circ = {e_2}^\circ$.

PROOF. Proof by structural induction on e. It is straightforward to verify that if e is a cvalue than e° is a value. Hence, we only need to focus on case (b).

The interesting case is application. Suppose we have $\Gamma \vdash_e e_1\ e_2 : \sigma$ and $e_1\ e_2 \longrightarrow e_3$ (in one step). Then either (a) e_1 can take a step (in which case the result follows by induction), or (b) e_1 is a cvalue. The latter has two sub-cases: either (b.1) e_1 is a plain value or (b.2) it is of form $(v_1 \blacktriangleright \gamma)$.

In case (b.1), since e can take a step, e_1 must be of form $\lambda x.e_1'$ so that $e_1\ e_2$ can take a (Beta) step. But then $(e_1\ e_2)^\circ$ can also take a (Beta) step. We need an auxiliary substitution lemma, that $[{e_2}^\circ/x]{e_1'}^\circ = [e_2/x]{e_1'}^\circ$, and then we are done.

In case (b.2), e_1 is of form $(v_1 \blacktriangleright \gamma)$, and by consistency v_1 must have a function type, and hence must be of the form $\lambda x.e_1'$. Hence $e_1\ e_2$ can take a (Push) step. Taking a (Push) step leaves the erasure of the term unchanged, modulo alpha conversion, which gives the result.

The other cases can be proved in a similar way. For example, suppose $\Gamma \vdash_e \textbf{case}\ e\ \textbf{of}\ \overline{p \rightarrow e} : \tau$. Then $\Gamma \vdash_e e : \sigma$ and $\overline{\Gamma \vdash_p p \rightarrow e : \sigma \rightarrow \tau}$. As before, the only interesting case is if e is a cvalue, otherwise, the result follows by induction. There are again two sub-cases to consider: (b.1) e is a plain value or (b.2) e is of the from $(v \blacktriangleright \gamma)$.

In case (b.1), e must be of the form $K\ \overline{\sigma}\ \overline{\varphi}\ \overline{e'}$, since the **case** expression can take a step. But then **case** e **of** $\overline{p \rightarrow e}^\circ$ can take a (Case) step and we are done.

In case (b.2), by consistency we find that e is of the form $K\ \overline{\sigma}\ \overline{\varphi}\ \overline{e'} \blacktriangleright \gamma$. Then, we can take a (KPush) step. This leaves the erasure of the term unchanged and we are done. □

COROLLARY 2 (Erasure soundness). *For an well-typed System* $\mathrm{F_C}$ *term* e_1*, we have* $e_1 \longrightarrow^* e_2$ *iff* ${e_1}^\circ \longrightarrow^* {e_2}^\circ$.

The dynamic semantics of Figure 4 makes all the coercions in the program bigger and bigger. This is not a run-time concern, because of erasure, but it might be a concern for compiler transformations. Fortunately there are many type-preserving simplifications that can be performed on coercions, such as:

$$\begin{array}{lcl} \mathrm{sym}\,\sigma &=& \sigma\\ \mathrm{left}\,(\mathit{Tree}\ \mathit{Int}) &=& \mathit{Tree}\\ e \blacktriangleright \sigma &=& e \end{array}$$

and so on. The compiler writer is free (but not obliged) to use such identities to keep the size of coercions under control.

In this context, it is interesting to note the connection of type-equality coercions to the notion of proof objects in machine-supported theorem proving. Coercion terms are essentially proof objects of equational logic and the above simplification rules, as well the manipulations performed by rules, such as (PushK), correspond to proof transformations.

3.10 Summary and observations

F_C is an intensional type theory, like F: that is, *every term encodes its own typing derivation*. This is bad for humans (because the terms are bigger) but good for a compiler (because type checking is simple, syntax-directed, and decidable). An obvious question is this: could we maintain simple, syntax-directed, decidable type-checking for F_C with less clutter? In particular, a coercion is an explicit proof of a type equality; could we omit the coercions, retaining only their kinds, and reconstructing the proofs on the fly?

No, we could not. Previous work showed that such proofs can indeed be inferred for the special case of GADTs [44, 35, 39]. But our setting is much more general because of our type functions, which in turn are necessary to support the source-language extensions we seek. Reconstructing an equality proof amounts to unification modulo an equational theory (E-unification), which is undecidable even in various restricted forms, let alone in the general case [2]. In short, dropping the explicit proofs encoded by coercions would render type checking undecidable (see Appendix B for a formal proof).

Why do we express coercions as *types*, rather than as *terms*? The latter is more conventional; for example, GADTs can be used to encode equality evidence [37], via a GADT of the form

```
data Eq a b where { EQ :: Eq a a }
```

F_C turns this idea on its head, instead using equality evidence to encode GADTs. This is good for several reasons. First, F_C is more foundational than System F plus GADTs. Second, F_C expresses equality evidence in *types*, which permit erasure; GADTs encode equality evidence as *values*, and these values cannot be erased. Why not? Because in the presence of recursion, the mere existence of an expression of type `Eq a b` is not enough to guarantee that `a` is the same as `b`, because $\perp$ has any type. Instead, one must *evaluate* evidence before using it, to ensure that it converges, or else provide a meta-level proof that asserts that the evidence always converges. In contrast, our language of types deliberately lacks recursion, and hence coercions can be trusted simply by virtue of being well-kinded.

4. Translating GADTs

With F_C in hand, we now sketch the translation of a source language supporting GADTs into F_C. As highlighted in §2.1, the key idea is to turn type equalities into coercion types. This approach strongly resembles the dictionary-passing translation known from translating type classes [17]. The difference is that we do not turn type equalities into values, rather, we turn them into types.

We do not have space to present a full source language supporting GADTs, but instead sketch its main features; other papers give full details [44, 10]. We assume that the GADT source language has the following syntax of types:

Polytypes	π	$\rightarrow$	$\eta \mid \forall a.\pi$
Constrained types	η	$\rightarrow$	$\tau \mid \tau \sim \tau \Rightarrow \eta$
Monotypes	τ	$\rightarrow$	$a \mid \tau \rightarrow \tau \mid T\,\overline{\tau}$

We deliberately re-use F_C's syntax $\tau_1 \sim \tau_2$ to describe GADT type equalities. These equality constraints are used in the source-language type of data constructors. For example, the `Succ` constructor from §2.1 would have type

$$\texttt{Succ} : \forall a.(a \sim \texttt{Int}) \Rightarrow \texttt{Int} \rightarrow \texttt{Exp}\ a$$

Notice that this already *is* an F_C type.

To keep the presentation simple, we use a non-syntax-directed translation scheme based on the judgement

$$C; \Gamma \vdash_{GADT} e : \pi \leadsto e'$$

We read it as "assuming constraint C and type environment Γ, the source-language expression e has type π, and translates to the F_C expression e'". The translation scheme can be made syntax-directed along the lines of [31, 35, 39]. The constraint C consists of a set of named type equalities:

$$C \quad \rightarrow \quad \epsilon \mid C, c\!:\!\tau_1 \sim \tau_2$$

The most interesting translation rules are shown in Figure 5, where we assume for simplicity that all quantified GADT variables are of kind $*$. The Rules (Var), ($\forall$-Intro), and ($\forall$-Elim), dealing with variables and the introduction and elimination of polymorphic types, are standard for translating Hindley/Milner to System F [19]. The introduction and elimination rules for constrained types, Rules (C-Intro) and (C-Elim), relate to the standard type-class translation [17], but where class constraints induce value abstraction and application, equality constraints induce type abstraction and application.

The translation of pattern clauses in Rule (Case) is as expected. We replace each GADT constructor by an appropriate F_C constructor which additionally carries coercion types representing the GADT type equalities. We assume that source patterns are already flat.

Rule (Eq) applies the cast construct to coerce types. For this, we need a coercion γ witnessing the equality of the two types, and we simply re-use the F_C judgement $\Gamma \vdash_{\text{co}} \gamma : \tau_1 \sim \tau_2$ from Figure 2. In this context, γ is an "output" of the judgement, a coercion whose syntactic structure describes the proof of $\tau_1 \sim \tau_2$. In other words, $C \vdash_{\text{co}} \gamma : \tau_1 \sim \tau_2$ represents the GADT condition that the equality context "C implies $\tau_1 \sim \tau_2$".

Finding a γ is decidable, using an algorithm inspired by the unification algorithm [23]. The key observation is that the statement "C implies $\tau_1 \sim \tau_2$" holds if $\theta(\tau_1) = \theta(\tau_2)$ where θ is the most general unifier of C. W.l.o.g., we neglect the case that C has no unifier, i.e. C is unsatisfiable. Program parts which make use of unsatisfiable constraints effectively represent dead-code.

Roughly, the type coercion construction procedure proceeds as follows. Given the assumption set C and our goal $\tau_1 \sim \tau_2$ we perform the following calculations:

Step 1 : We normalise the constraints $C = \overline{c : \tau' \sim \tau''}$ to the *solved* form $\overline{\gamma : a \sim v}$ where $a_i < a_{i+1}$ and $fv(\bar{a}) \cap fv(\bar{v}) = \emptyset$ by decomposing with Rule (Right) (we neglect higher-kinded types for simplicity) and applying Rule (Sym) and (Trans). We assume some suitable ordering among variables with $<$ and disallow recursive types.

Step 2 : Normalise $c' : \tau_1 \sim \tau_2$ where c' is fresh to the solved form $\overline{\gamma' : a' \sim v'}$ where $a'_j < a'_{j+1}$.

Step 3 : Match the resulting equations from Step 2 against equations from Step 1.

Step 4 : We obtain γ by reversing the normalisation steps in Step 2.

Failure in any of the steps implies that $C \vdash_{\text{co}} \gamma : \tau_1 \sim \tau_2$ does not hold for any γ. A constraint-based formulation of the above algorithm is given in [40].

To illustrate the algorithm, let's consider $C = \{c_1 : [a] \sim [b], c_2 : b = c\}$ and $c_3 : [a] \sim [c]$, with $a < b < c$.
Step 1: Normalising C yields $\{\text{right}\ c_1 : a \sim b, c_2 : b = c\}$ in an intermediate step. We apply rule (Trans) to obtain the solved form $\{(\text{right}\ c_1) \circ c_2 : a \sim c, c_2 : b = c\}$
Step 2: Normalising $c_3 : [a] \sim [c]$ yields $(\text{right}\ c_3) : a \sim c$.
Step 3: We can match right $c_3 : a \sim c$ against $(\text{right}\ c_1 \circ c_2) : a \sim c$.
Step 4: Reversing the normalisation steps in Step 2 yields $c_3 = [\text{right}\ c_1 \circ c_2]$, as $\vdash_{\text{co}} [\,] : [\,] \sim [\,]$.

The following result can be straightforwardly proven by induction over the derivation.

$$\boxed{C;\Gamma \vdash_{GADT} e : \pi \leadsto e'}$$

$$\text{(Var)}\quad \frac{(x:\pi) \in \Gamma}{C;\Gamma \vdash_{GADT} x : \pi \leadsto x} \qquad \text{(Eq)}\quad \frac{C;\Gamma \vdash_{GADT} e : \tau \leadsto e' \quad C \vdash_{\text{co}} \gamma : \tau \sim \tau'}{C;\Gamma \vdash_{GADT} e : \tau' \leadsto e' \blacktriangleright \gamma}$$

$$(\forall\text{-Intro})\quad \frac{C;\Gamma \vdash_{GADT} e : \pi \leadsto e' \quad a \notin fv(C,\Gamma)}{C;\Gamma \vdash_{GADT} e : \forall a.\pi \leadsto \Lambda a{:}{*}.\, e'} \qquad \text{(C-Intro)}\quad \frac{C, c{:}\tau_1 \sim \tau_2;\ \Gamma \vdash_{GADT} e : \eta \leadsto e'}{C;\Gamma \vdash_{GADT} e : \tau_1 \sim \tau_2 \Rightarrow \eta \leadsto \Lambda(c{:}\tau_1 \sim \tau_2).\, e'}$$

$$(\forall\text{-Elim})\quad \frac{C;\Gamma \vdash_{GADT} e : \forall a.\pi \leadsto e'}{C;\Gamma \vdash_{GADT} e : [\tau/a]\pi \leadsto e'\ \tau} \qquad \text{(C-Elim)}\quad \frac{C;\Gamma \vdash_{GADT} e : \tau_1 \sim \tau_2 \Rightarrow \eta \leadsto e' \quad C \vdash_{\text{co}} \gamma : \tau_1 \sim \tau_2}{C;\Gamma \vdash_{GADT} e : \eta \leadsto e'\ \gamma}$$

$$\boxed{C;\Gamma \vdash_{GADT} p \to e : \pi \to \pi \leadsto p' \to e'}$$

$$\text{(Alt)}\quad \frac{\begin{array}{c} K :: \forall \bar{a}, \bar{b}. \overline{\tau' \sim \tau''} \Rightarrow \overline{\tau} \to T\ \bar{a} \quad \bar{a} \cap \bar{b} = \emptyset \quad fv(\overline{\tau}, \overline{\tau'}, \overline{\tau''}) = fv(\bar{a}, \bar{b}) \quad \theta = [\overline{\upsilon/a}] \\ C, \overline{c{:}\theta(\tau') \sim \theta(\tau'')};\ \Gamma, \overline{x{:}\theta(\tau)} \vdash_{GADT} e : \tau' \leadsto e' \quad \bar{c}\ \text{fresh} \end{array}}{C;\Gamma \vdash_{GADT} K\ \overline{x} \to e : T\ \overline{\upsilon} \to \tau' \leadsto K\ (\overline{b{:}{*}})\ (\overline{c{:}\theta(\tau') \sim \theta(\tau'')})\ (\overline{x{:}\theta(\tau)}) \to e'}$$

Figure 5: Type-Directed GADT to F_C Translation (interesting cases)

LEMMA 1 (Type Preservation). *Let $C;\emptyset \vdash_{GADT} e : t \leadsto e'$. Then, $C \vdash_e e' : t$.*

In §3.5, we saw that only consistent F_C programs are sound. It is not hard to show that this the case for GADT F_C programs, as GADT programs only make use of syntactic (a.k.a. Herbrand) type equality, and so, require no type functions at all.

THEOREM 4 (GADT Consistency). *If $dom(\Gamma)$ contains no type variables or coercion constants, and $\Gamma \vdash_{\text{co}} \gamma : \sigma_1 \sim \sigma_2$, then $\sigma_1 = \sigma_2$ (i.e. the two are syntactically identical).*

The proof is by induction on the structure of γ. Consistency is an immediate corollary of Theorem 4. Hence, all GADT F_C programs are sound. From the Erasure Soundness Corollary 2, we can immediately conclude that the semantics of GADT programs remains unchanged (where e° is e after type erasure).

LEMMA 2. *Let $\emptyset;\emptyset \vdash_{GADT} e : t \leadsto e'$. Then, $e' \rightarrowtail^* v$ iff $e^\circ \rightarrowtail^* v$ where v is some base value, e.g. integer constants.*

5. Translating Associated Types

In §2.2, we claimed that F_C permits a more direct and more general type-preserving translation of associated types than the translation to plain System F described in [6]. In fact, the translation of associated types to F_C is almost embarrassingly simple, especially given the translation of GADTs to F_C from §4. In the following, we outline the additions required to the standard translation of type classes to System F [17] to support associated types.

5.1 Translating expressions

To translate expressions, we need to add three rules to the standard system of [17], namely Rules (Eq), (C-Intro), and (C-Elim) from Figure 5 of the GADT translation. Rule (Eq) permits casting expression with types including associated types to equal types where the associated types have been replaced by their definition. Strictly speaking, the Rules (C-Intro) and (C-Elim) are used in a more general setting during associated type translation than during GADT translation. Firstly, the set C contains not only equalities, but both equality and class constraints. Secondly, in the GADT translation only GADT data constructors carry equality constraints, whereas in the associated type translation, any function can carry equality constraints.

5.2 Translating class predicates

In the standard translation, predicates are translated to dictionaries by a judgement $C \Vdash_D D\ \tau \leadsto \nu$. In the presence of associated types, we have to handle the case where the type argument to a predicate contains an associated type. For example, given the class

```
class Collects c where
  type Elem c       -- associated type synonym
  empty  :: c
  insert :: Elem c -> c -> c
  toList :: c -> [Elem c]
```

we might want to define

```
sumColl :: (Collects c, Num (Elem c))
        => c -> Elem c
sumColl c = sum (toList c)
```

which sums the elements of a collection, provided these elements are members of the `Num` class; i.e., provided we have `Num (Elem c)`. Here we have an associated type as a parameter to a class constraint. Wherever the function $sumColl$ is used, we will have to check the constraint `Num (Elem c)`, which will require a cast of the resulting dictionary if c is instantiated. We achieve this by adding the following rule:

$$\text{(Subst)}\quad \frac{C \Vdash_D D\ \tau_1 \leadsto w \qquad C \vdash_{\text{TY}} \gamma : D\ \tau_1 = D\ \tau_2}{C \Vdash_D D\ \tau_2 \leadsto w \blacktriangleright \gamma}$$

It permits to replace type class arguments by equal types, where the coercion γ witnessing the equality is used to adapt the type of the dictionary w, which in turn witnesses the type class instance. Interestingly, we need this rule also for the translation as soon as we admit qualified constructor signatures in GADT declarations.

5.3 Translating declarations

Strictly speaking, we also have to extend the translation rules for class and instance definitions, as these can now declare and define associated types. However, the extension is so small that we omit the formal rules for space reasons. In summary, each declaration of an associated type in a type class turns into the declaration of a type function in F_C, and each definition of an associated type in an instance turns into an equality **axiom** in F_C. We have seen examples of this in §2.2.

5.4 Observations

In the translation of associated types, it becomes clear why F_C includes coercions over type constructors of higher kind. Consider the following class of monads with references:

```
class Monad m => RefMonad m where
  type Ref m :: * -> *
  newRef :: a -> m (Ref m a)
  readRef :: Ref m a -> m a
  writeRef :: Ref m a -> a -> m ()
```

This class may be instantiated for the `IO` monad and the `ST` monad. The associated type `Ref` is of higher-kind, which implies that the coercions generated from its definitions will also be higher kinded.

The translation of associated types to plain System F imposes two restrictions on the formation of well-formed programs [5, §5.1], namely (1) that equality constraints for an n parameter type class must have type variables as the first n arguments to its associated types and (2) that class method signatures cannot constrain type class parameters. Both constraints can be lifted in the translation to F_C.

5.5 Guaranteeing consistency for associated types

How do we know that the axioms generated by the source-program associated types and their instance declarations are consistent? The answer is simple. The source-language type system for associated types only makes sense if the instance declarations obey certain constraints, such as non-overlap [6]. Under those conditions, it is easy to guarantee that the axioms derived from the source program are consistent. In this section we briefly sketch why this is the case.

The axiom generated by an instance declaration for an associated type has the form[1] $C\ :\ (\forall \overline{a:\star}.S\ \sigma_1) \sim (\forall \overline{a:\star}.\sigma_2)$. where (a) σ_1 does not refer to any type function, (b) $fv(\sigma_1) = \bar{a}$, and (c) $fv(\sigma_2) \subseteq \bar{a}$. This is an entirely natural condition and can also be found in [5]. We call an axiom of this form a *rewrite axiom*, and a set of such axioms defines a rewrite system among types.

Now, the source language rules ensure that this rewrite system is *confluent* and *terminating*, using the standard meaning of these terms [2]. We write $\sigma_1 \downarrow \sigma_2$ to mean that σ_1 can be rewritten to σ_2 by zero or more steps, where σ_2 is a normal form. Then we prove that each type has a canonical normal form:

THEOREM 5 (Canonical Normal Forms). *Let Γ be well-formed, terminating and confluent. Then, $\Gamma \vdash_{\text{co}} \gamma : \sigma_1 \sim \sigma_2$ iff $\sigma_1 \downarrow \sigma_1'$ and $\sigma_2 \downarrow \sigma_2'$ such that $\sigma_1' = \sigma_2'$.*

Using this result we can decide type equality via a canonical normal form test, and thereby prove consistency:

COROLLARY 3 (AT Consistency). *If Γ contains only rewrite axioms that together form a terminating and confluent rewrite system, then Γ is consistent.*

For example, assume $\Gamma \vdash_{\text{co}} \gamma : T_1\ \overline{\sigma_1} \sim T_2\ \overline{\sigma_2}$. Then, we find $T_1\ \overline{\sigma_1} \downarrow \sigma_1'$ and $T_2\ \overline{\sigma_2} \downarrow \sigma_2'$ such that $\sigma_1' = \sigma_2'$. None of the rewrite rules affect T_1 or T_2. Hence, σ_1' must have the shape $T_1\ \overline{\sigma_1''}$ and σ_2' the shape $T_2\ \overline{\sigma_2''}$. Immediately, we find that $T_1 = T_2$ and we are done.

We can state similar results for type functions resulting from functional dependencies. Again, the canonical normal form property is the key to obtain consistency. While sufficient the canonical normal form property is not a necessary condition. Consider the non-confluent but consistent environment $\Gamma = \{c_1 : S_1\ [Int] \sim S_2, c_2 : (\forall a{:}\star.S_1\ [a]) \sim (\forall a{:}\star.[S_1\ a])\}$. We find that $\Gamma \vdash_{\text{co}} \gamma : S_1\ [Int] \sim S_2$. But there exists $S_1\ [Int] \downarrow [S_1\ Int]$ and $S_2 \downarrow S_2$ where $[S_1\ Int] \neq S_2$. Similar observations can be made for ill-formed, consistent environments.

[1] For simplicity, we here assume unary associated types that do not range over higher-kinded types.

6. Related Work

System F with GADTs. Xi et al. [44] introduced the explicitly typed calculus $\lambda_{2,G\mu}$ together with a translation from an implicitly typed source language supporting GADTs. Their calculus has the typing rules for GADTs built in, just like Pottier & Régis-Gianas's MLGI [35]. This is the approach that GHC initially took. F_C is the result of a search for an alternative.

Encoding GADTs in plain System F and F_ω. There are several previous works [3, 9, 30, 43, 7, 40] which attempt an encoding of GADTs in plain System F with (boxed) existential types. We believe that these primitive encoding schemes are not practical and often non-trivial to achieve. We discuss this in more detail in Appendix A.

An encoding of a restricted subset of GADT programs in plain System F_ω can be found in [33], but this encoding only works for limited patterns of recursion.

Intentional type analysis and beyond. Harper and Morrisett's visionary paper on intensional type analysis [20] introduced the calculus λ_i^{ML}, which was already sufficiently expressive for a large range of GADT programs, although GADTs only became popular later. Subsequently, Crary and Weirch's language LX [12] generalised the approach significantly by enabling the analysis of source language types in the intermediate language and by providing a type erasure semantics, among other things. LX's type analysis is sufficiently powerful to express *closed* type functions which must be primitive recursive. This is related, but different to F_C(X), where type functions are *open* and need not be terminating (see also Appendix B).

Trifonov et al. [42] generalised λ_i^{ML} in a different direction than LX, such that they arrived at a fully reflexive calculus; i.e., one that can analyse the type of any runtime value of the calculus. In particular, they can analyse types of higher kind, an ability that was also crucial in the design of F_C(X). However, Trifonov et al.'s work corresponds to λ_i^{ML} and LX in that it applies to *closed*, primitive-recursive type functions.

Calculi with explicit proofs. Licata & Harper [25] introduced the calculus LH to represent programs in the style of Dependent ML. LH's type terms include lambdas, and its definitional equality therefore includes a beta rule, whereas F_C's definitional equality is simpler, being purely syntactic. LH's propositional equality enables explicit proofs of type equality, much as in F_C(X). These explicit proofs are the basis for the definition of *retyping functions* that play a similar role to our cast expressions. In contrast, FC's propositional equality lacks some of LH's equalities, namely those including certain forms of inductive proofs as well as type equalities whose retypings have a computational effect. The price for LH's added expressiveness is that retypings — even if they amount to the identity on values — can incur non-trivial runtime costs and (together with LH types) cannot be erased without meta-level proofs that assert that particular forms of retypings are guaranteed to be identity functions.

Another significant difference is that in LH, as in LX, type functions are *closed* and must be *primitive recursive*; whereas in F_C(X), they are open and need not be terminating. These properties are very important in our intended applications, as we argued in Section 3.4. Finally, F_C(X) admits optimising transformations that are not valid in LH, as we discussed in Section 3.8.

Shao et al.'s impressive work [36] illustrates how to integrate an entire proof system into typed intermediate and assembly languages, such that program transformations preserve proofs. Their type language TL resembles the calculus of inductive constructions (CIC) and, among other things, can express retypings witnessed by explicit proofs of equality [36, Section 4.4], not unlike LH. TL is much more expressive and complex than $F_C(X)$ and, like LH, does not support open type functions.

Coercion-based subtyping. Mitchell [29] introduced the idea of inserting coercions during type inference for an ML-like languages. However, Mitchell's coercion are not identities, but perform coercions between different numeric types and so forth. A more recent proposal of the same idea was presented by Kießling and Luo [22]. Subsequently, Mitchell [28] also studied coercions that are operationally identities to model type refinement for type inference in systems that go beyond Hindley/Milner.

Much closer to F_C is the work by Breazu-Tannen et al. [4] who add a notion of coercions to System F to translate languages featuring inheritance polymorphism. In contrast to F_C, their coercions model a subsumption relationship, and hence are not symmetric. Moreover, their coercions are values, not types. Nevertheless, they introduce coercion combinators, as we do, but they don't consider decomposition, which is crucial to translating GADTs. The focus of their paper is the translation of an extended version of Cardelli & Wegner's Fun, and in particular, the coherence properties of that translation.

Similarly, Crary [11] introduces a coercion calculus for inclusive subtyping. It shares the distinction between plain values and coercion values with our system, but does not require quantification over coercions, nor does it consider decomposition.

Intuitionistic type theory, dependent types, and theorem provers. The ideas from Mitchell's work [29, 28] have also been transferred to dependently typed calculi as they are used in theorem provers; e.g., based on the Calculus of Constructions [8]. Generally, our coercion terms are a simple instance of the proof terms of logical frameworks, such as LF [18], or generally the evidence in intuitionistic type theory [26]. This connection indicates several directions for extending the presented system in the direction of more powerful dependently typed languages, such as Epigram [27].

Translucency and singleton kinds. In the work on ML-style module systems, type equalities are represented as singleton kinds, which are essential to model translucent signatures [14]. Recent work [15] demonstrated that such a module calculus can represent a wide range of type class programs including associated types. Hence, there is clearly an overlap with $F_C(X)$ equality axioms, which we use to represent associated types. Nevertheless, the current formulation of modular type classes covers only a subset of the type class programs supported by Haskell systems, such as GHC. We leave a detailed comparison of the two approaches to future work.

7. Conclusions and further work

We showed that explicit evidence for type equalities is a convenient mechanism for the type-preserving translation of GADTs, associative types, and functional dependencies. We implemented $F_C(X)$ in its full glory in GHC, a widely used, state-of-the-art, highly optimising Haskell compiler. At the same time, we re-implemented GHC's support for `newtype`s and GADTs to work as outlined in §2 and added support for associated (data) types. Consequently, this implementation instantiates the decision procedure for consistency, "X", to a combination of that described in Section 4 and 5. The F_C-version of GHC is now *the* main development version of GHC and supports our claim that $F_C(X)$ is a practical choice for a production system.

An interesting avenue for future work is to find good source language features to expose more of the power of F_C to programmers.

Acknowledgements

We thank James Cheney, Karl Crary, Roman Leshchinskiy, Dan Licata, Conor McBride, Benjamin Pierce, François Pottier, Robert Harper, and referees for ICFP'06, POPL'07 and TLDI'07 for their helpful comments on previous versions of this paper. We are grateful to Stephanie Weirich for interesting discussions during the genesis of F_C, to Lennart Augustsson for a discussion on encoding associated types in System F, and to Andreas Abel for pointing out the connection to [33].

References

[1] M. Abadi, L. Cardelli, and P.-L. Curien. Formal parametric polymorphism. In *Proc. of POPL '93*, pages 157–170, New York, NY, USA, 1993. ACM Press.

[2] F. Baader and T. Nipkow. *Term rewriting and all that*. Cambridge University Press, 1999.

[3] A. I. Baars and S. D. Swierstra. Typing dynamic typing. In *Proc. of ICF'02*, pages 157–166. ACM Press, 2002.

[4] V. Breazu-Tannen, T. Coquand, C. Gunter, and A. Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93(1):172–221, July 1991.

[5] M. M. T. Chakravarty, G. Keller, and S. Peyton Jones. Associated type synonyms. In *Proc. of ICFP '05*, pages 241–253, New York, NY, USA, 2005. ACM Press.

[6] M. M. T. Chakravarty, G. Keller, S. Peyton Jones, and S. Marlow. Associated types with class. In *Proc. of POPL '05*, pages 1–13. ACM Press, 2005.

[7] C. Chen, D. Zhu, and H. Xi. Implementing cut elimination: A case study of simulating dependent types in Haskell. In *Proc. of PADL'04*, volume 3057 of *LNCS*, pages 239–254. Springer-Verlag, 2004.

[8] G. Chen. Coercive subtyping for the calculus of constructions. In *Proc. of POPL'03*, pages 150–159, New York, NY, USA, 2003. ACM Press.

[9] J. Cheney and R. Hinze. A lightweight implementation of generics and dynamics. In *Proc. of Haskell Workshop'02*, pages 90–104. ACM Press, 2002.

[10] J. Cheney and R. Hinze. First-class phantom types. TR 1901, Cornell University, 2003.

[11] K. Crary. Typed compilation of inclusive subtyping. In *Proc. of ICFP'00*, pages 68–81, New York, NY, USA, 2000. ACM Press.

[12] K. Crary and S. Weirich. Flexible type analysis. In *Proc. of ICFP'99*, pages 233–248. ACM Press, 1999.

[13] N. Dershowitz and J.-P. Jouannaud. *Handbook of Theoretical Computer Science (Volume B: Formal Models and Semantics)*, chapter 6: Rewrite Systems. Elsevier Science Publishers, 1990.

[14] D. Dreyer, K. Crary, and R. Harper. A type system for higher-order modules. In *Proc. of POPL'03*, pages 236–249, New York, NY, USA, 2003. ACM Press.

[15] D. Dreyer, R. Harper, and M. M. T. Chakravarty. Modular type classes. In *Proc. of POPL '07*. ACM Press, 2007. To appear.

[16] G. J. Duck, S. Peyton Jones, P. J. Stuckey, and M. Sulzmann. Sound and decidable type inference for functional dependencies. In *Proc. of (ESOP'04)*, number 2986 in LNCS, pages 49–63. Springer-Verlag, 2004.

[17] C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler. Type classes in Haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, 1996.

[18] R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. In *Proc. of LICS'87*, pages 194–204. IEEE Computer Society Press, 1987.

[19] R. Harper and J. C. Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252, 1993.

[20] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Proc. of POPL'95*, pages 130–141. ACM Press, 1995.

[21] M. P. Jones. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming (ESOP 2000)*, number 1782 in Lecture Notes in Computer Science. Springer-Verlag, 2000.

[22] R. Kießling and Z. Luo. Coercions in Hindley-Milner systems. In *Types for Proofs and Programs: Third International Workshop, TYPES 2003*, number 3085 in LNCS, pages 259–275, 2004.

[23] J. Lassez, M. Maher, and K. Marriott. Unification revisited. In *Foundations of Deductive Databases and Logic Programming*. Morgan Kauffman, 1987.

[24] K. Läufer and M. Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems*, 16(5):1411–1430, 1994.

[25] D. R. Licata and R. Harper. A formulation of Dependent ML with explicit equality proofs. Technical Report CMU-CS-05-178, Carnegie Mellon University, Dec. 2005.

[26] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis·Napoli, 1984.

[27] C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[28] J. Mitchell. Polymorphic type inference and containment. In *Logical Foundations of Functional Programming*, pages 153–193. Addison-Wesley, 1990.

[29] J. C. Mitchell. Coercion and type inference. In *Proc of POPL'84*, pages 175–185. ACM Press, 1984.

[30] E. Pasalic. *The Role of Type Equality in Meta-Programming*. PhD thesis, Oregon Health & Science University, OGI School of Science & Engineering, September 2004.

[31] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *Proc. of ICFP'06*, pages 50–61. ACM Press, 2006.

[32] B. Pierce. *Types and programming languages*. MIT Press, 2002.

[33] B. Pierce, S. Dietzen, and S. Michaylov. Programming in higher-order typed lambda-calculi. Technical report, Carnegie Mellon University, 1989.

[34] E. L. Post. Recursive unsolvability of a problem of Thue. *Journal of Symbolic Logic*, 12:1—1, 1947.

[35] F. Pottier and Y. Régis-Gianas. Stratified type inference for generalized algebraic data types. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–244. ACM Press, 2006.

[36] Z. Shao, V. Trifonov, B. Saha, and N. Papaspyrou. A type system for certified binaries. *ACM Trans. Program. Lang. Syst.*, 27(1):1–45, 2005.

[37] 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, New York, NY, USA, 2004. ACM Press.

[38] M. Sulzmann, G. J. Duck, S. Peyton Jones, and P. J. Stuckey. Understanding functional dependencies via constraint handling rules. *Journal of Functional Programming*, 2006. To appear.

[39] M. Sulzmann, T. Schrijvers, and P. J. Stuckey. Type inference for GADTs via Herbrand constraint abduction. `http://www.comp.nus.edu.sg/~sulzmann`, July 2006.

[40] M. Sulzmann and M. Wang. A systematic translation of guarded recursive data types to existential types. Technical Report TR22/04, The National University of Singapore, 2004.

[41] M. Sulzmann, J. Wazny, and P. J. Stuckey. A framework for extended algebraic data types. In *Proc. of FLOPS'06*, volume 3945 of *LNCS*, pages 47–64. Springer-Verlag, 2006.

[42] V. Trifonov, B. Saha, and Z. Shao. Fully reflexive intensional type analysis. In *Proc. of ICFP'00*, pages 82–93, New York, NY, USA, 2000. ACM Press.

[43] S. Weirich. Type-safe cast (functional pearl). In *Proc. of ICFP'00*, pages 58–67. ACM Press, 2000.

[44] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *Proc. of POPL '03*, pages 224–235, New York, NY, USA, 2003. ACM Press.

A. Primitive Translation of GADTs

We attempt a primitive translation (encoding) of GADTs to System F with (boxed) existential types (for convenience we will use Haskell extended with rank-n types and existentials). We provide evidence that such an encoding is sometimes hard to achieve.

The gist of the primitive encoding idea is to model type equality $a \sim b$ via safe coercion functions. Effectively, a pair of embedding/projection functions. Each type cast $\gamma \blacktriangleright e$ is then turned into the function application $\gamma\ e$. To ensure correctness of this encoding scheme, we need to guarantee that at run-time each coercion γ evaluates to the identity.

There are two approaches known in the literature to encode such coercion functions. One approach, employed in [3, 9, 30, 43], uses "Leibniz" equality

```
newtype EQ a b =
   Proof { apply :: forall f . f a -> f b }
refl :: EQ a a
refl    =  Proof id
newtype Flip f a b =  Flip { unFlip :: f b a }
symm :: EQ a b -> EQ b a
symm p =  unFlip (apply p (Flip refl))
trans :: EQ a b -> EQ b c -> EQ a c
trans p q =  Proof (apply q . apply p)
newtype List f a =  List { unList :: f [a] }
list :: EQ a b  -> EQ [a] [b]
list p =  Proof (unList . apply p . List)
```

We also provide a few sample type coercion functions. As pointed out in [7], the trouble with this approach is that it seems impossible to define "decomposition" functions such as

```
decompList :: EQ [a] [b] -> EQ a b
```

The alternative method is to represent type equality as follows.

```
type EQ a b = (a->b,b->a)
refl :: EQ a a
refl = (id,id)
sym :: EQ a b -> EQ b a
sym (f,g) = (g,f)
trans :: EQ a b -> EQ b c -> EQ a c
trans (f1,g1) (f2,g2) = (f2.f1,g1.g2)
list :: EQ a b -> EQ [a] [b]
list (f,g) = (map f, map g)
```

The advantage is that decomposition is possible for some types but not for all as will see at the end of this section. Though, many (if not all) realistic GADT programs can be translated based on this encoding [40]. On the other hand, the (serious) disadvantage of this representation is that it may incur a severe run-time penalty. Consider the definition of `list` where we have to apply the coercion functions to each element.

Let's attempt an encoding of the trie example found in [10]. A trie is a finite map from keys to values whose structure depends on the type of keys, here encoded as products and sums in GADT variants:

```
data Either a b where
   Left  :: a -> Either a b
   Right :: b -> Either a b
data Trie k v where
```

```
  TUnit ::
    Maybe v                 -> Trie () v
  TSum  :: forall k1 k2.
    Trie k1 v -> Trie k2 v -> Trie (Either k1 k2) v
  TProd :: forall k1 k2.
    Trie k1 (Trie k2 v)    -> Trie (k1, k2) v
```

A trie for a unit type is maybe one value, a trie for a sum is a product of tries, and a trie for a product is a composition of tries. An important operation on tries is the merging of two maps with the same domain and co-domain.

```
merge :: (v -> v -> v)
      -> Trie k v -> Trie k v -> Trie k v
merge c (TUnit Nothing ) (TUnit Nothing  ) =
  TUnit Nothing
merge c (TUnit Nothing ) (TUnit (Just v')) =
  TUnit (Just v')
merge c (TUnit (Just v)) (TUnit Nothing  ) =
  TUnit (Just v)
merge c (TUnit (Just v)) (TUnit (Just v')) =
  TUnit (Just (c v v'))
merge c (TSum ta tb)     (TSum ta' tb')    =
  TSum (merge c ta ta') (merge c tb tb')
merge c (TProd ta)       (TProd ta')       =
  TProd (merge (merge c) ta ta')
```

The second two last equations are interesting. The patterns of the first and second argument constrain `k` to `Either k1 k2` and `Either k1' k2'`, respectively. Hence, we have

$$\texttt{Either k1 k2} = \texttt{k} = \texttt{Either k1}' \texttt{ k2}'$$

from which we can follow $\texttt{k1} = \texttt{k1}'$ and $\texttt{k2} = \texttt{k2}'$. The point is that to translate the above to $\mathrm{F_C}$, we need to construct a coercion that witness these equalities, we need decomposition.

To encode the trie example, we need (among others) a function

```
decomp :: EQ (Either a b) (Either c d) -> EQ a b
```

But it seems impossible to define such a function if we use Leibniz equality.

Let's consider the "other" type equality representation. To ensure correctness of the encoding scheme, we need to maintain the invariant that for any type coercion function `coerce :: EQ a b -> EQ c d` we have that `coerce` applied to a pair of identity functions yields another pair of identity functions. We are more lucky here, a function `decomp:: EQ (Either a b) (Either c d) -> EQ a c` with the above property is actually definable.

For simplicity, we only give parts of the definition of `decomp`.

```
decomp1 :: (Either a b -> Either c d) -> (a->c)
decomp1 f  = \ a -> case (f (Left a)) of
                       Left c -> c
```

We inject the `a` value into the `Either` data type, apply the incoming coercing function and then extract the `c` value. It is easy to verify that the invariant is satisfied.

There are many other examples which can be translated using the "other" type equality representation [40]. In fact, it almost seems that all practical examples can be encoded. Though, not every decomposition function is definable. Here is the (contrived) critical example.

```
data Foo a  where
   K :: Foo a
data Erk a b c where
   I :: c -> Erk a a c
f :: Erk (Foo a) (Foo Int) a -> a
f (I x) = x + 1
```

$$(\mathrm{Cast}_i)\quad \frac{\Gamma \vdash_e e : \sigma \quad \Gamma \vdash_{\mathrm{CO}} \gamma : \sigma \sim \tau}{\Gamma \vdash_e (e \blacktriangleright \{\sigma \sim \tau\}) \rightsquigarrow (e \blacktriangleright \gamma) : \tau}$$

$$(\mathrm{AppT_{TY}})\quad \frac{\Gamma \vdash_e e : \forall a{:}\kappa.\sigma \quad \Gamma \vdash_k \kappa : \mathrm{TY} \quad \Gamma \vdash_{\mathrm{TY}} \tau : \kappa}{\Gamma \vdash_e (e\, \tau) \rightsquigarrow (e\, \tau) : \sigma[\tau/a]}$$

$$(\mathrm{AppT_{CO}})\quad \frac{\Gamma \vdash_e e : \forall a{:}(\tau \sim \upsilon).\sigma \qquad \Gamma \vdash_k (\tau \sim \upsilon) : \mathrm{CO} \quad \Gamma \vdash_{\mathrm{CO}} \varphi : \tau \sim \upsilon}{\Gamma \vdash_e (e\, \{\tau \sim \upsilon\}) \rightsquigarrow (e\, \varphi) : \sigma[\varphi/a]}$$

Figure 6: Modified typing rules for System $\mathrm{F}_{\mathrm{C}_i}$

First, we convince ourselves that the above program is well-typed. The pattern `I x` in combination with the type annotation implies that $\texttt{Foo a} = \texttt{Foo Int}$. By decomposition, we conclude that $\texttt{a} = \texttt{Int}$. Thus, the program text `x + 1` can be given type `Int`. Hence, the above is well-typed. To translate the above, we need to define a function of type `EQ (Foo a) (Foo Int) -> EQ a Int`. We claim it is impossible to define such a function with satisfies the invariant. It suffices to show that a function

```
decompFoo :: (Foo a->Foo Int)->(a->Int)
```

with the property that `decompFoo ( x->x)` evaluates to `x->x` is not definable.

The problem here is that a value of type `a` cannot be injected into a value of type `Foo a`. So, clearly the incoming function of type `Foo a->Foo Int` is useless. Effectively, we could omit the function parameter altogether. Parametricity tells us that any function of type `a->Int` must be a constant function. Hence, `decompFoo` applied to any function of type `Foo a->Foo Int` yields a constant function. Hence, an encoding of the above critical example is impossible.

In fact, the "decomposition" problem is hardly surprising given that similar issues arise when translating type class programs [17].

```
class Foo a where foo :: a->Int
instance Foo a => Foo [a] where
    foo [] = 1
    foo _  = 2
bar :: Foo [a] => a->Int
bar = foo
```

Based on the System F-style translation scheme described in [17], we are unable to translate function `bar`. The program text demands a dictionary for `Foo a` but the annotation only supplies a dictionary for `Foo [a]`. This is the wrong way around. The instance declaration tells us how to construct `Foo [a]` given `Foo a` but the other direction does not hold in general.

B. Complexity of Type Checking

Previous calculi for GADTs, such as $\lambda_{2,G\mu}$ [44] and MLGX [35], did not pass evidence for coercions explicitly, but deduced the equality between types at coercion points implicitly during type checking. We call such calculi *calculi with implicit evidence.* This raises the question whether it is necessary to construct and pass evidence explicitly in $\mathrm{F_C}$, or whether we could not have made it into an implicit calculus. To answer this question, we define an implicit variant of $\mathrm{F_C}$, which we call $\mathrm{F}_{\mathrm{C}_i}$ and show that type checking for $\mathrm{F}_{\mathrm{C}_i}$ is undecidable. More precisely, we show that reconstructing explicit coercion terms, which amount to proofs justifying coercions, is undecidable for $\mathrm{F}_{\mathrm{C}_i}$.

The difference between $\mathrm{F_C}$ and $\mathrm{F}_{\mathrm{C}_i}$ is simply the following: wherever $\mathrm{F_C}$ has a coercion type γ of kind $\sigma_1 \sim \sigma_2$, $\mathrm{F}_{\mathrm{C}_i}$ only gives the equality kind in curly braces; i.e., $\{\sigma_1 \sim \sigma_2\}$. Hence,

- casts $e \blacktriangleright \gamma$ turn into $e \blacktriangleright \{\sigma_1 \sim \sigma_2\}$ and
- type applications $e\ \gamma$ turn into $e\ \{\sigma_1 \sim \sigma_2\}$.

It's obviously straight forward to turn an $\mathrm{F_C}$ program into an $\mathrm{F_{C_i}}$ program. The converse, recovering an $\mathrm{F_C}$ program from $\mathrm{F_{C_i}}$, requires a type-directed translation, that we obtain from the typing rules of Figure 2 by turning the expression typing rules into translation rules. We replace the Rules (AppT) and (Cast) by those in Figure 6; for all other rules, the translation is the identity. The modified Rules (Cast$_i$) and (AppT$_{\mathrm{CO}}$) use the judgement $\Gamma \vdash_{\mathrm{CO}} \gamma : \sigma \sim \tau$ to re-compute γ. As we will see next, computing γ from a kind $\sigma \sim \tau$ is, in the general case, undecidable.

THEOREM 6 (Undecidability of coercion reconstruction in $\mathrm{F_{C_i}}$). *Given an environment* Γ *and an* $\mathrm{F_{C_i}}$ *expression* e*, computing the corresponding* $\mathrm{F_C}$ *expression* e' *and its type* σ *as determined by* $\Gamma \vdash_e e \leadsto e' : \sigma$ *is not decidable.*

PROOF. We show that the reconstruction of coercion types for $\mathrm{F_{C_i}}$ expressions includes the word problem for A-ground theories, which is long known to be undecidable [34]. An A-ground theory is defined over a signature $\mathcal{F}$ including the binary symbol $Plus$ and a set of $\mathcal{F}$-equations E that are all ground (i.e, variable-free), except for the associativity of $Plus$. More concretely, we have

$$\begin{aligned}\mathcal{F} = \{&S_1 : \overline{\star}^{k_1} \to \star,\ \ldots,\ S_n : \overline{\star}^{k_n} \to \star,\\ &Plus : \star \to \star \to \star\}\end{aligned}$$

where $\overline{\star}^{k} \to \star$ indicates that S_i is k-ary. Furthermore, we have

$$\begin{aligned}E = \{&\sigma_1 = \tau_1,\ \ldots,\ \sigma_m = \tau_m,\\ &Plus\ (Plus\ a\ b)\ c) = Plus\ a\ (Plus\ b\ c))\}\end{aligned}$$

where the σ_i and τ_i are terms over $\mathcal{F}$.

We represent $\mathcal{F}$ and E in $\mathrm{F_C}$'s type language as follows:

$$\begin{aligned}
&\mathbf{type}\ S_1\ :\ \overline{\star}^{k_1} \to \star\\
&\quad\vdots\\
&\mathbf{type}\ S_n\ :\ \overline{\star}^{k_n} \to \star\\
&\mathbf{type}\ Plus : \star \to \star \to \star\\
&\mathbf{data}\ Term\ :\ \star \to \star\ \mathbf{where}\\
&\quad Sv_1\quad : \forall \overline{a}^{k_1}.\ \overline{Nat\ a}^{k_1}\ \to\ Nat\ (S_1\ \overline{a}^{k_1})\\
&\quad\vdots\\
&\quad Sv_n\quad : \forall \overline{a}^{k_n}.\ \overline{Nat\ a}^{k_1}\ \to\ Nat\ (S_n\ \overline{a}^{k_n})\\
&\quad Plusv : \forall a\ b.\ Nat\ a\ \to\ Nat\ b\ \to\ Nat\ (Plus\ a\ b)\\
&\mathbf{axiom}\ ax_1\ :\ \sigma_1 = \tau_1\\
&\quad\vdots\\
&\mathbf{axiom}\ ax_m\ :\ \sigma_m = \tau_m\\
&\mathbf{axiom}\ assoc\ :\\
&\quad (\forall a\ b\ c.\ Plus\ (Plus\ a\ b)\ c) \sim\ (\forall a\ b\ c.\ Plus\ a\ (Plus\ b\ c))
\end{aligned}$$

The data type $Term$ enables us to construct any (ground) $\mathcal{F}$-term by reflection from the structurally identical $\mathrm{F_C}$ expression using $Term$'s constructors. For example, if S_1 and S_2 are nullary, we have that $Plusv\ Sv_1\ Sv_2\ :\ Term\ (Plus\ S_1\ S_2)$. If σ is an $\mathcal{F}$-term, we denote the structurally identical $\mathrm{F_C}$ expression with $\widehat{\sigma}$ and have $\widehat{\sigma}\ :\ Term\ \sigma$.

The word problem for the A-ground theory E over the signature $\mathcal{F}$ amounts to testing for two arbitrary $\mathcal{F}$-terms σ and τ whether $\sigma = \tau$ under E. We represent this as an $\mathrm{F_{C_i}}$ type checking problem by typing the cast expression $\widehat{\sigma} \blacktriangleright \{\sigma \sim \tau\}$ in the context of the above $\mathrm{F_C}$ declarations corresponding to $\mathcal{F}$ and E. The undecidability of the word problem implies the undecidability of $\mathrm{F_{C_i}}$ typing, or more precisely, that the judgement $\Gamma \vdash_{\mathrm{CO}} \gamma : \sigma \sim \tau$ in the premise of $\mathrm{F_{C_i}}$'s Rule (Cast$_i$) cannot be realised by an effective decision procedure when γ is unknown. □

It remains the question whether there exists a restriction on $\mathrm{F_{C_i}}$ equality axioms that excludes encoding problems, such as the word problem for A-ground theories, but is still sufficient for translating GADTs, associated types, functional dependencies, and so forth. Given the range of FD programs supported by GHC and the analysis of properties of FD programs in [38], this is not a viable approach.