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-0614
August 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) 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. F_C can therefore serve as a typed intermediate language in a compiler that supports these features.

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 $\mathrm{F_C}$, which extends System F with support for non-syntactic type equality. There are two main extensions: (i) explicit witnesses for type equalities, and (ii) non-parametric type functions, given meaning by top-level equality axioms. Unlike System F, $\mathrm{F_C}$ 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. $\mathrm{F_C}$ can therefore serve as a typed intermediate language in a compiler that supports these features.

1. Introduction

GHC (the Glasgow Haskell Compiler) has a problem. It uses System F, extended with algebraic data types and case-expressions, as its typed intermediate language. But Haskell has incubated a variety of extensions that are hard to translate into this language: in particular, functional dependencies [18], generalised algebraic data types (GADTs) [39, 27], and associated types [6, 5]. When we added GADTs to GHC, we also 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, the translation became extremely clumsy and, in some interesting corner cases, impossible.

In this paper we resolve this dilemma by presenting System $\mathrm{F_C}$, a superset of F that is both *more foundational* and *more powerful* than *ad hoc* extensions such as GADTs or associated types. $\mathrm{F_C}$ 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. Our specific contributions are these:

- We articulate the problem in §2, and give an informal description of our solution.
- We give a formal description of System $\mathrm{F_C}$, 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).
- The system is very general, but we are able to identify *consistency* as a necessary and sufficient property for soundness (§3.4). Conditions identified in earlier work on GADTs, associated types, and functional dependencies, are special cases of consistency.
- As one application, we give a type-preserving translation of a source language supporting GADTs into System $\mathrm{F_C}$ (§4). As a second, we sketch a similar translation for associated types (§5). The latter, and the corresponding translation for functional dependencies, are more general than all previous type-preserving translations for these features.

We have implemented $\mathrm{F_C}$ in GHC. Although the implementation is not yet complete, we have done enough to convince ourselves that $\mathrm{F_C}$ can be incorporated in the guts of a complicated compiler without great upheaval.

System $\mathrm{F_C}$ 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. We discuss this and other related work in §6.

Although we have written this introduction as if GHC were the only compiler in the world, there is nothing GHC-specific about System $\mathrm{F_C}$. Indeed, $\mathrm{F_C}$ occupies an interesting place in the design space. On the one hand, we know of no language smaller than $\mathrm{F_C}$ for which typing is decidable and that is sufficient to serve as a target for GADTs, associated types, and functional dependencies (see also Appendix B and C); while on the other, $\mathrm{F_C}$ seems much more general than any of these specific source-language constructs, much as System F is far more general than ML. In fact, we believe that $\mathrm{F_C}$ may have much wider application as a typed target for sophisticated HOT (higher-order typed) source languages — a hypothesis that remains to be tested.

2. The key ideas

The polymorphic lambda calculus, System F, is popular as a highly-expressive typed intermediate language in compilers for functional languages. Many source-language features can be desugared into System F with little difficulty, but not all. The first such feature 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. Whilst these can be encoded in System F, the encoding is heavy, and compilers invariably extend System F by adding algebraic data types, data constructors, and **case** expressions. We will use $\mathrm{F_A}$ to describe System F extended in this way, where the data constructors are allowed to have existential components [21], type variables can be of higher kind, and type constructor applications can be partial.

* Also Computer Science Department, Boston University, kevind@bu.edu

Over the last few years, source languages (notably Haskell) have started to explore language features that embody *non-syntactic type equality*. These features include functional dependencies [14], generalised algebraic data types (GADTs) [39, 33], 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 $\mathrm{F_C}$, an extension of System F that resolves the dilemma. We show how it can serve as a target for each of the three examples, and we sketch further possibilities (such as closed type functions) that $\mathrm{F_C}$ opens up. The formal details are presented in §3. Throughout we use `typewriter font` for source-code, and $italics$ for $\mathrm{F_C}$.

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 current "solution" — we instead propose a general mechanism of parameterising functions with *type equalities* witnessed by *coercion types*, which we call *type equality 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:

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

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 [39, 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:

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

Here a ranges over *types*, of kind $\star$, while co ranges over *coercions*, of kind $a \doteq \mathit{Int}$. An important property of our approach is that *coercions are types*, and hence, *equalities* $\tau_1 \doteq \tau_2$ *are dependent kinds.* An equality kind $\tau_1 \doteq \tau_2$ categorises all coercion types that witness the equality of the two types t_1 and t_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, $\mathit{Int}\ :\ (\mathit{Int} \doteq \mathit{Int})$ is an instance of the reflexivity of equality and $\mathrm{sym}\ \mathit{co}\ :\ (\mathit{Int} \doteq a)$, with $\mathit{co}\ :\ (a \doteq \mathit{Int})$, 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:

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

We use $@$ to distinguish type, and hence also coercion, application from value application. 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 $\mathrm{F_C}$:

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

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 $(\mathrm{sym}\ co)$ to get a coercion from Int to a and use that to cast the type of 0 to a, using the *cast expression* $0 \blacktriangleright \mathrm{sym}\ co$. 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 $g\ :\ (t_1 \doteq t_2)$. 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.

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 [15] 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 $\mathit{negate}\ :\ \mathit{NumDict}\ a\ \to\ a\ \to\ a$, 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 $\mathrm{F_C}$:

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

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 [e]` instance:

$$\begin{array}{l}\mathbf{axiom}\ elemList\ :\ (\forall e{:}\star.Elem\ [e]) \doteq (\forall e{:}\star.e)\\ dCollectsList\ :\ \forall e{:}\star.EqDict\ e\ \rightarrow\ CollectsDict\ [e]\\ dCollectsList\ =\ ...\end{array}$$

The **axiom** definition introduces a new, named *coercion constant*, $elemList$, which serves as a witness of the equality asserted by the axiom; here, that we can convert between types of the form $Elem\ [e]$ and e, for any $\star$-kinded e. Using this coercion, we can $insert$ the character 'b' into a list of characters ['a'] by applying the instantiated coercion $elemList\ @\ Char$ backwards to 'b', thus:

$$(\text{'b'} \blacktriangleright \mathrm{sym}\ (elemList\ @\ Char)) : Elem\ [Char]$$

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 at all, even clumsily, and $\mathrm{F_C}$ supports them too, as we sketch in §5.

2.3 Functional dependencies

Functional dependencies are another popular extension of Haskell's type-class mechanism [18]. With functional dependencies, we can encode a function over types `F` as a relation; i.e., the dependent type parameters are not unlike associated types:

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

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)
```

Then 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:

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

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. (When constructing the witnesses d_1 and d_2 we use the coercion combinators $\mathrm{sym}\ \cdot$ and $\cdot \rhd \cdot$, which represent symmetry and transitivity, respectively. Moreover, we lift the coercion d_1 to function space by applying the type constructor $\rightarrow$, which is admissible as plain types and type constructors witness reflexivity.)

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 `newtype`, and closed type functions

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

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

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 $\mathrm{F_C}$ allows it to be handled directly, by introducing a new axiom

$$\mathbf{axiom}\ CoT\ :\ (T \rightarrow T) \doteq T$$

More ambitiously, we may consider closed type functions, as provided by (for example) Ωmega [33]. (Associated types constitute *open*, that is extensible, type functions.) To review the basic

idea, consider the following definition of bounded sequences whose length is encoded in their type:

```
kind Nat = Z | S Nat

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

We define bounded sequences with a GADT parametrised by a kind `Nat` encoding Peano numerals with two new type constructors `Z :: Nat` and `S :: Nat -> Nat`. Now, what is the signature of a function appending one such sequence at another? The length of the resulting sequence is clearly the sum of the lengths of the two component sequences. Hence, we want to write something along the following lines:

```
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)
```

Here `Plus` is a closed type function:

```
type Plus Z     b = b
     Plus (S a) b = S (Plus a b)
```

The type function `Plus` can be directly translated into $\mathrm{F_C}$; its equations become equality axioms; and the GADT `Seq` is encoded as in §2.1. $\mathrm{F_C}$ does not support the declaration of algebraic data kinds, such as `Nat`, but that is an orthogonal extension that could easily be added. (Or, perhaps better, one could re-use types as kinds, but that is another story.) In a similar manner, the recently proposed extended algebraic data types [37], which add equality and predicate constraints to GADTs, can be translated to $\mathrm{F_C}$.

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 $\mathrm{F_C}$ as a solution to these difficulties. We will formalise it in the next section.

3. System $\mathrm{F_C}$

The main idea in $\mathrm{F_C}$ is that we pass around explicit evidence for type equalities, in just the same way that System F passes types explicitly. Indeed, in $\mathrm{F_C}$ 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 §3.7.

Figure 1 defines the syntax of System $\mathrm{F_C}$, while Figure 2 gives 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 maybe an expression, type term, or environment.

3.1 Conventional features

System $\mathrm{F_C}$ is a superset of System F. The syntax of types and kinds is given in Figure 1. Like F, $\mathrm{F_C}$ 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 human reader; formally, the coercion typing and kinding rules enforce the appropriate restrictions. Like $F\omega$ (and Haskell), our system allows types of higher kind; hence the type application form $\tau_1\ \tau_2$.

Symbol Classes

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

Declarations

$$
\begin{array}{lcl}
pgm & \rightarrow & \overline{decl};\ e \\
decl & \rightarrow & \mathbf{data}\ T : \overline{\kappa} \rightarrow \star\ \mathbf{where} \\
 & & \quad \overline{K : \forall \overline{a:\kappa}.\, \forall \overline{b:\iota}.\, \overline{\sigma} \rightarrow T\ \overline{a}} \\
 & \mid & \mathbf{type}\ S_n : \overline{\kappa}^n \rightarrow \iota \\
 & \mid & \mathbf{axiom}\ C : \sigma_1 \doteq \sigma_2
\end{array}
$$

Sorts and kinds

δ	$\rightarrow$	TY ∣ CO	Sorts
κ, ι	$\rightarrow$	$\star \mid \kappa_1 \rightarrow \kappa_2 \mid \sigma_1 \doteq \sigma_2$	Kinds

Types and Coercions

d	$\rightarrow$	$a \mid T$	Atom of sort TY
g	$\rightarrow$	$c \mid C$	Atom of sort CO
$\varphi, \rho, \sigma, \tau, \upsilon, \gamma$	$\rightarrow$	$a \mid C \mid T \mid \varphi_1\ \varphi_2 \mid S_n\ \overline{\varphi}^n \mid \forall a:\kappa.\, \varphi$	
	$\mid$	$\mathrm{sym}\,\gamma \mid \gamma_1 \rhd \gamma_2 \mid \gamma\ @\ \varphi \mid \mathrm{left}\,\gamma \mid \mathrm{right}\,\gamma$	

(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	$\rightarrow$	$x \mid K$	Variables and data constructors
e	$\rightarrow$	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
	$\mid$	$\mathbf{let}\ x:\sigma = e_1\ \mathbf{in}\ e_2$	
	$\mid$	$\mathbf{case}\ e_1\ \mathbf{of}\ \overline{p \rightarrow e_2}$	
	$\mid$	$e \blacktriangleright \gamma$	Cast
p	$\rightarrow$	$K\ \overline{b:\kappa}\ \overline{x:\sigma}$	Pattern

Environments

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

Figure 1: Syntax of System $\mathrm{F_C}$

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 $\sigma_1 \rightarrow \sigma_2$ as the curried application of a built-in function type constructor to two arguments, thus $(\rightarrow)\ \sigma_1\ \sigma_2$. Furthermore, although we give the syntax of arrow types and quantified types in an uncurried way, we also sometimes use the following syntactic sugar:

$$
\begin{array}{rcl}
\overline{\varphi}^n \rightarrow \varphi_r & \equiv & \varphi_1 \rightarrow \cdots \rightarrow \varphi_n \rightarrow \varphi_r \\
\forall \overline{\alpha}^n.\varphi & \equiv & \forall \alpha_1 \cdots \forall \alpha_n.\varphi
\end{array}
$$

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

$$K : \forall \overline{a:\kappa.}^n \forall \overline{b:\iota}.\, \overline{\varphi} \rightarrow T\ \overline{a}^n$$

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

$$\text{(Star)}\ \frac{}{\Gamma \vdash_k \star : \mathrm{TY}} \qquad \text{(FunK)}\ \frac{\Gamma \vdash_k \kappa_1 : \mathrm{TY} \quad \Gamma \vdash_k \kappa_2 : \mathrm{TY}}{\Gamma \vdash_k \kappa_1 \to \kappa_2 : \mathrm{TY}} \qquad \text{(EqTy)}\ \frac{\Gamma \vdash_{\mathrm{TY}} \sigma_1 : \kappa \quad \Gamma \vdash_{\mathrm{TY}} \sigma_2 : \kappa}{\Gamma \vdash_k \sigma_1 \doteq \sigma_2 : \mathrm{CO}}$$

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

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

$$\text{(SConT)}\ \frac{(S_n : \overline{\kappa}^n \to \iota) \in \Gamma \quad \Gamma \vdash_{\mathrm{TY}} \overline{\sigma : \kappa}^n}{\Gamma \vdash_{\mathrm{TY}} S_n\ \overline{\sigma}^n : \iota} \qquad \text{(AllT)}\ \frac{\Gamma, a : \kappa \vdash_{\mathrm{TY}} \sigma : \star \quad \Gamma \vdash_k \kappa : \delta \quad a \notin \mathit{fv}(\Gamma)}{\Gamma \vdash_{\mathrm{TY}} \forall a{:}\kappa_1.\, \sigma : \star}$$

$\boxed{\Gamma \vdash_{\mathrm{CO}} \gamma : \sigma \doteq \tau}$

$$\text{(Refl)}\ \frac{a : \kappa \in \Gamma \quad \Gamma \vdash_k \kappa : \mathrm{TY}}{\Gamma \vdash_{\mathrm{CO}} a : a \doteq a} \qquad \text{(AtomC)}\ \frac{g : \sigma \doteq \tau \in \Gamma \quad \Gamma \vdash_{\mathrm{TY}} \sigma : \kappa}{\Gamma \vdash_{\mathrm{CO}} g : \sigma \doteq \tau}$$

$$\text{(InstT)}\ \frac{\Gamma \vdash_{\mathrm{CO}} \gamma : \forall a{:}\kappa.\, \sigma \doteq \forall b{:}\kappa.\, \tau \quad \Gamma \vdash_k \kappa : \mathrm{TY} \quad \Gamma \vdash_{\mathrm{TY}} \upsilon : \kappa}{\Gamma \vdash_{\mathrm{CO}} \gamma\,@\,\upsilon : [\upsilon/a]\sigma \doteq [\upsilon/b]\tau} \qquad \text{(InstC)}\ \frac{\Gamma \vdash_{\mathrm{CO}} \gamma_1 : (\tau_1 \doteq \tau_2 \Rightarrow \sigma_1) \doteq (\upsilon_1 \doteq \upsilon_2 \Rightarrow \sigma_2) \quad \Gamma \vdash_{\mathrm{CO}} \gamma_2 : \upsilon_1 \doteq \upsilon_2}{\Gamma \vdash_{\mathrm{CO}} \gamma_1\,@\,\gamma_2 : \sigma_1 \doteq \sigma_2}$$

$$\text{(AllT)}\ \frac{\Gamma, a : \kappa \vdash_{\mathrm{CO}} \gamma : \sigma \doteq \tau \quad \Gamma \vdash_k \kappa : \mathrm{TY} \quad a \notin \mathit{fv}(\Gamma)}{\Gamma \vdash_{\mathrm{CO}} \forall a{:}\kappa.\, \gamma : \forall a{:}\kappa.\, \sigma \doteq \forall a{:}\kappa.\, \tau} \qquad \text{(AllC)}\ \frac{\Gamma \vdash_{\mathrm{CO}} \gamma_1 : \tau_1 \doteq \upsilon_1 \quad \Gamma \vdash_{\mathrm{CO}} \gamma_2 : \tau_2 \doteq \upsilon_2 \quad \Gamma \vdash_k \tau_1 \doteq \tau_2 : \mathrm{CO} \quad \Gamma \vdash_{\mathrm{CO}} \gamma : \sigma_1 \doteq \sigma_2}{\Gamma \vdash_{\mathrm{CO}} (\gamma_1 \doteq \gamma_2) \Rightarrow \gamma : ((\tau_1 \doteq \tau_2) \Rightarrow \sigma_1) \doteq ((\upsilon_1 \doteq \upsilon_2) \Rightarrow \sigma_2)}$$

$$\text{(Comp)}\ \frac{\Gamma \vdash_{\mathrm{CO}} \gamma_1 : \sigma_1 \doteq \tau_1 \quad \Gamma \vdash_{\mathrm{CO}} \gamma_2 : \sigma_2 \doteq \tau_2 \quad \Gamma \vdash_{\mathrm{TY}} \sigma_1\ \sigma_2 : \kappa}{\Gamma \vdash_{\mathrm{CO}} \gamma_1\ \gamma_2 : \sigma_1\ \sigma_2 \doteq \tau_1\ \tau_2} \qquad \text{(Sym)}\ \frac{\Gamma \vdash_{\mathrm{CO}} \gamma : \sigma \doteq \tau}{\Gamma \vdash_{\mathrm{CO}} \mathrm{sym}\,\gamma : \tau \doteq \sigma} \qquad \text{(Trans)}\ \frac{\Gamma \vdash_{\mathrm{CO}} \gamma_1 : \sigma_1 \doteq \sigma_2 \quad \Gamma \vdash_{\mathrm{CO}} \gamma_2 : \sigma_2 \doteq \sigma_3}{\Gamma \vdash_{\mathrm{CO}} \gamma_1 \rhd \gamma_2 : \sigma_1 \doteq \sigma_3}$$

$$\text{(SComp)}\ \frac{\Gamma \vdash_{\mathrm{CO}} \overline{\gamma : \sigma \doteq \tau}^n \quad \Gamma \vdash_{\mathrm{TY}} S_n\ \overline{\sigma}^n : \kappa}{\Gamma \vdash_{\mathrm{CO}} S_n\ \overline{\gamma}^n : S_n\ \overline{\sigma}^n \doteq S_n\ \overline{\tau}^n} \qquad \text{(DLeft)}\ \frac{\Gamma \vdash_{\mathrm{CO}} \gamma : \sigma_1\ \sigma_2 \doteq \tau_1\ \tau_2}{\Gamma \vdash_{\mathrm{CO}} \mathrm{left}\,\gamma : \sigma_1 \doteq \tau_1} \qquad \text{(DRight)}\ \frac{\Gamma \vdash_{\mathrm{CO}} \gamma : \sigma_1\ \sigma_2 \doteq \tau_1\ \tau_2}{\Gamma \vdash_{\mathrm{CO}} \mathrm{right}\,\gamma : \sigma_2 \doteq \tau_2}$$

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

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

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

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

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

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

$\boxed{\Gamma \vdash \mathit{decl} : \Gamma'}$

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

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

$\boxed{\Gamma \vdash \mathit{pgm} : \sigma}$

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

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

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 $\sigma_1 \doteq \sigma_2$, the structure of kinds 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 (AppT). Finally, the rules for judgements of the form $\Gamma \vdash_k \kappa : \delta$ ensure the well-formedness of kinds; we will return to the meaning of δ when discussing the kinds of coercion types.

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

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

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 $\mathrm{F_C}$ 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 $\mathrm{F_C}$, a coercion, γ, is a special sort of type whose kind takes the unusual form $\sigma_1 \doteq \sigma_2$. This kind indicates that γ is a proof that the types σ_1 and σ_2 are equal.

More concretely, we can use a coercion $\gamma : (\sigma_1 \doteq \sigma_2)$, to cast an expression $e : \sigma_1$ to type σ_2 using the *cast expression* $(e \blacktriangleright \gamma)$ (see Rule (Cast) in Figure 2). The term syntax for type abstraction $\Lambda a.e$ and application $e\,@\,\varphi$ also serves for coercion abstraction and application.

Coercions are types, but they have their own kinding judgement $\vdash_{\text{CO}}$, given in Figure 2. The type of a term often has the form $\forall co{:}(\sigma_1 \doteq \sigma_2).\tau$, where τ does not mention co — indeed, the kind system would reject any mention of co in τ. We allow the standard syntactic sugar for this case, writing it thus: $(\sigma_1 \doteq \sigma_2) \Rightarrow \tau$ (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 '$\mathrm{sym}\,\gamma$', transitivity '$\gamma_1 \rhd \gamma_2$', type composition '$\gamma_1\ \gamma_2$', and decomposition '$\mathrm{left}\,\gamma$' and '$\mathrm{right}\,\gamma$'. 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

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

(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 $Int : (Int \doteq Int)$ by Rule (Refl). Rule (Refl) 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 $Exp\ a$:

$$\begin{array}{l} foo\ :\ \forall\ a{:}\star.\ Exp\ a\ \to\ a\ \to\ a \\ \quad =\ \Lambda a{:}\star.\, \lambda e{:}Exp\ a.\, \lambda x{:}a. \\ \quad\quad \mathbf{case}\ e\ \mathbf{of} \\ \quad\quad\quad Succ\ (co{:}a \doteq Int)\ (e'{:}Exp\ Int)\ \to \\ \quad\quad\quad\quad (foo\,@\,Int\ e'\ 0\ +\ (x \blacktriangleright co)) \blacktriangleright \mathrm{sym}\ co \end{array}$$

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 $(\mathrm{sym}\,co)$.

Coercion composition allows us to "lift" coercions through arbitrary types. For example, if we have a coercion $\gamma{:}\sigma_1 \doteq \sigma_2$ then the coercion $Tree\ \gamma$ is evidence that $Tree\ \sigma_1 \doteq Tree\ \sigma_2$. More generally, if $\gamma : \sigma_1 \doteq \sigma_2$, then

$$[\gamma/a]\varphi\ :\ [\sigma_1/a]\varphi \doteq [\sigma_2/a]\varphi$$

for any type φ, including polytypes. As a more elaborate example,

$$\forall b.\gamma \to Int\ :\ (\forall b.\sigma_1 \to Int) \doteq (\forall b.\sigma_2 \to Int)$$

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

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* $\overline{\kappa}^n \to \iota$, 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 (§3.4 explains why). The arity subscript should be considered part of the name of the type constructor, although we will often elide it, writing $Elem\ \sigma$ rather than $Elem_1\ \sigma$, for example.

A type function receives its interpretation by one or more equality **axiom** definitions. Each axiom introduces a coercion constant, whose kind specifies the types between which the coercion converts. For example,

$$\mathbf{axiom}\ elemBitSet\ :\ Elem\ BitSet \doteq Char$$

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

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

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

To use this axiom as a coercion, say, for lists of integers, we need to apply the coercion constant to a type argument:

$$elemList\,@\,Int\ :\ (Elem\ \ [Int] \doteq Int)$$

which appeals to Rule (InstT) of Figure 2. The introduction rule corresponding to the elimination Rule (InstT) is Rule (AllT). These introduction and elimination rules for parametrisation over types of

terms is mirrored in the coercion world by Rules (InstC) and (AllC). (Remember that $\kappa \Rightarrow \sigma$ is sugar for $\forall \_ : \kappa.\, \sigma$.) The corresponding coercions are needed when a coercion is applied to a GADT in the scrutinee of a **case** expression.

It may be surprising that we use one quantifier on each side of the equality, instead of quantifying over the entire equality as in

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

The advantage of the notation used in the definition of $\mathit{elemList}$ is that we do not need quantifiers as part of the kind structure, which simplifies matters significantly. Moreover, this notation is justified by the logical structure of parametricity as captured in Abadi et al.'s logical relations between parametric types [1]. A well known instance of such a logical relation is the subsumption relation between parametric types, of which type equality is the symmetric closure. For example, we obviously have $(\forall b.\forall a.a \to b) \leq (\forall b.\mathit{Int} \to b)$.

3.4 Consistency and saturation

In System $\mathrm{F_C}$, 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:

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

(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 one is this: we require that the axioms, taken together, are *consistent*. We essentially adapt the standard notion of consistency of sets of equations [12, Section 3] to our setting.

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

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

1. *If $\Gamma \vdash_{\mathrm{CO}} \gamma : T\ \overline{\sigma} \doteq \upsilon$, and υ is a value type, then $\upsilon = T\ \overline{\tau}$.*
2. *If $\Gamma \vdash_{\mathrm{CO}} \gamma : \forall a{:}\kappa.\sigma \doteq \upsilon$, 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 $\mathit{Bool} \doteq \mathit{Int}$. It is clear that the condition is necessary for soundness, and it turns out (§3.5) that it is also sufficient. Rule (Pgm) in Figure 2 enforces consistency.

Consistency is both necessary and sufficient for soundness of $\mathrm{F_C}$, but it is not obvious how to check whether a particular program is consistent. For particular classes of programs, however, consistency is easy to guarantee, as we will see for the $\mathrm{F_C}$ programs generated from GADTs (in §4) and associated types (in §5).

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:

$$\mathbf{axiom}\ c1\ :\ S_1\ \mathit{Int} \doteq \mathit{Bool}$$
$$\mathbf{axiom}\ c2\ :\ S_1\ \mathit{Bool} \doteq \mathit{Bool}$$

Here, both $S_1\ \mathit{Int}$ and $S_1\ \mathit{Char}$ are represented by the Bool type. But now we can form the coercion $(c1 \rhd (\mathrm{sym}\, c2))$ which has type $S_1\ \mathit{Int} \doteq S_1\ \mathit{Bool}$, and from that we must not be able to deduce (via right) that $\mathit{Int} \doteq \mathit{Bool}$, 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 (DLeft) and (DRight) 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:

$$\mathbf{type}\ \mathit{HS}_1\ :\ \star \to \star \to \star$$
$$\mathbf{axiom}\ c1\ :\ \mathit{HS}_1\ \mathit{Int} \doteq [\,]$$
$$\mathbf{axiom}\ c2\ :\ \mathit{HS}_1\ \mathit{Bool} \doteq \mathit{Maybe}$$

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.

3.5 Dynamic semantics and soundness

The operational semantics of $\mathrm{F_C}$ is shown in Figure 3. 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 3. 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 3). The latter form is needed because we cannot reduce a term to a plain value without losing type preservation; for example, we cannot reduce $(\mathit{True} \blacktriangleright \mathit{co})$, where $\mathit{co} : \mathit{Bool} \doteq S$ any further without changing its type from S to Bool.

However, there are three situations when a cvalue will not do, namely as the function part of a type application or function application, and as the scrutinee of a **case** expression. Rules (TPush), (Push) and (PushC) deal with those three situations, by pushing the coercion inside the term, turning the cast into a plain value. Notice that all three 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, but (PushC) is more complicated. Here is an example, stripped of the **case** context, where $\mathit{Cons} : \forall a.a \to [a] \to [a]$, and $\gamma : [\mathit{Int}] \doteq [S\ \mathit{Bool}]$:

$$\mathit{Cons}\ \mathit{Int}\ e_1\ e_2 \blacktriangleright \gamma \longrightarrow \mathit{Cons}\ (S\ \mathit{Bool})\ \ (e_1 \blacktriangleright \mathrm{right}\,\gamma)\ \ (e_2 \blacktriangleright ([\,])\ (\mathrm{right}\,\gamma))$$

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.)

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.

Notice that 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.

THEOREM 1 (Progress). *Suppose that 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.

Values:

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

Evaluation contexts:

$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']} \qquad E ::= [\,] \mid E\ e \mid E\ \tau \mid E \blacktriangleright \gamma$$

Expression reductions:

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

$$\begin{array}{lll} (TPush) & ((\Lambda a{:}\kappa.\, e) \blacktriangleright \gamma)\,@\,\varphi & \longrightarrow (\Lambda a{:}\kappa'.\, (e \blacktriangleright \gamma\,@\,a))\,@\,\varphi \\ & & \text{where}\ \ \gamma : \forall b{:}\kappa.\, \sigma_1 \doteq \forall c{:}\kappa'.\, \sigma_2 \\ (Push) & ((\lambda x.e) \blacktriangleright \gamma)\ e' & \longrightarrow (\lambda y.([(y \blacktriangleright \gamma_1)/x]e \blacktriangleright \gamma_2))\ e' \\ & & \text{where}\ \ \gamma_1 = \mathrm{sym}\,(\mathrm{right}\,(\mathrm{left}\,\gamma)) \quad \text{– coercion for argument} \\ & & \qquad\ \ \gamma_2 = \mathrm{right}\,\gamma \quad \text{– coercion for result} \end{array}$$

$$\begin{array}{lll} (PushC) & \mathbf{case}\ (K\ \overline{\sigma}\ \overline{\varphi}\ \overline{e} \blacktriangleright \gamma)\ \mathbf{of}\ \overline{p \to rhs} & \longrightarrow \mathbf{case}\ (K\ \overline{\tau}\ \overline{\varphi'}\ \overline{e'})\ \mathbf{of}\ \overline{p \to rhs} \\ & & \text{where}\ \ \gamma : T\ \overline{\sigma} \doteq T\ \overline{\tau} \\ & & \qquad K : \forall \overline{a{:}\kappa}.\, \forall \overline{b{:}\iota}.\, \overline{\rho} \to T\ \overline{a} \\ & & \qquad \varphi'_i = \begin{cases} \mathrm{sym}\,\theta(\upsilon_1) \rhd \varphi_i \rhd \theta(\upsilon_2) & \text{if } b_i : \upsilon_1 \doteq \upsilon_2 \\ \varphi_i & \text{otherwise} \end{cases} \\ & & \qquad e'_i = e_i \blacktriangleright \theta(\rho_i) \\ & & \qquad \theta = [\overline{\gamma_i/a_i}, \overline{\varphi_i/b_i}] \\ & & \qquad \gamma_i = \mathrm{right}\,(\underbrace{(\mathrm{left} \ldots (\mathrm{left}}_{i-1}\ \gamma)) \end{array}$$

Figure 3: Operational semantics

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 $\Gamma \vdash_e \mathbf{case}\ e\ \mathbf{of}\ \overline{p \to e} : \tau$. Then $\Gamma \vdash_e e : \sigma$ and $\overline{\Gamma \vdash_p p \to e : \sigma \to \tau}$. 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 $K\ \overline{e'}$. Hence we can take a (Case) step (we assume that case expressions have exhaustive alternatives).
3. e is $v \blacktriangleright \gamma$. By consistency and since v is a value, v must be of form $K\ \overline{e'}$, so we can take a type-preserving step using (PushC).

□

COROLLARY 1 (Syntactic Soundness). *Let Γ be consistent 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.

3.6 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 (see Appendix A):

THEOREM 2. *Given $\Gamma \vdash_e e_1 : \sigma$, (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 ${e_1}^\circ = {e_2}^\circ$.*

The dynamic semantics of Figure 3 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 (PushL), correspond to proof transformations.

3.7 Summary and observations

$\mathrm{F_C}$ 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 $\mathrm{F_C}$ 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 [39, 31]. 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 C for a formal proof).

Why do we express coercions as *types*, rather than as *terms*? Doing so is more conventional; for example, GADTs can be used to encode equality evidence [33, §9], via a GADT of the form

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

$\mathrm{F_C}$ turns this idea on its head, instead using equality evidence to encode GADTs. This is good for several reasons. First, $\mathrm{F_C}$ is more foundational than System F plus GADTs. Second, $\mathrm{F_C}$ 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 $\bot$ has any type. Instead, one must *evaluate* evidence before using it, to ensure that it 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 $\mathrm{F_C}$ in hand, we now sketch the translation of a source language supporting GADTs into $\mathrm{F_C}$. 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 [15]. 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 [39, 10]. We assume that the GADT source language has the following syntax of types:

$$\begin{array}{lrcl} \text{Polytypes} & \pi & \rightarrow & \eta \mid \forall a.\pi \\ \text{Constrained types} & \eta & \rightarrow & t \mid t \doteq t \Rightarrow \eta \\ \text{Monotypes} & \tau & \rightarrow & a \mid \tau \rightarrow \tau \mid T\ \overline{\tau} \end{array}$$

We deliberately re-use $\mathrm{F_C}$'s syntax $\tau_1 \doteq \tau_2$ 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

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

Notice that this already *is* an $\mathrm{F_C}$ type.

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

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

We read it as "assuming constraint C and type environment Γ, the source-language expression e has type π, and translates to the $\mathrm{F_C}$ expression e'". The translation scheme can be made syntax-directed along the lines of [27, 31, 35]. The constraint C consists of a set of named type equalities:

$$C \quad \rightarrow \quad \epsilon \mid C, c{:}\tau_1 \doteq \tau_2$$

The most interesting translation rules are shown in Figure 4, where we assume for simplicity that all quantified GADT variables are of kind $*$. The Rules (Var), (∀-Intro), and (∀-Elim), dealing with variables and the introduction and elimination of polymorphic types, are standard for translating Hindley/Milner to System F [17]. The introduction and elimination rules for constrained types, Rules (C-Intro) and (C-Elim), relate to the standard type-class translation [15], 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 $\mathrm{F_C}$ 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 $\mathrm{F_C}$ judgement $\Gamma \vdash_{\mathrm{co}} \gamma : \tau_1 \doteq \tau_2$ from Figure 2. In this context, γ is an "output" of the judgement, a coercion whose syntactic structure describes the proof of $\tau_1 \doteq \tau_2$. In other words, $C \vdash_{\mathrm{co}} \gamma : \tau_1 \doteq \tau_2$ represents the GADT condition that the equality context "C implies $\tau_1 \doteq \tau_2$".

Finding a γ is decidable, using an algorithm inspired by the unification algorithm [20]. The key observation is that the statement "C implies $\tau_1 \doteq \tau_2$" 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 $\tau_1 \doteq \tau_2$ we perform the following calculations:

Step 1 : We normalise the constraints $C = \overline{c : \tau' \doteq \tau''}$ to the *solved* form $\overline{\gamma : a \doteq \upsilon}$ where $a_i < a_{i+1}$ and $\mathit{fv}(\bar{a}) \cap \mathit{fv}(\bar{\upsilon}) = \emptyset$ by decomposing with Rule (DRight) (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 $c' : \tau_1 \doteq \tau_2$ where c' is fresh to the solved form $\overline{\gamma' : a' \doteq \upsilon'}$ 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 $C \vdash_{\mathrm{co}} \gamma : \tau_1 \doteq \tau_2$ does not hold for any γ. A constraint-based formulation of the above algorithm is given in [36].

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

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

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

In §3.4, we saw that only consistent $\mathrm{F_C}$ programs are sound. It is not hard to show that this the case for GADT $\mathrm{F_C}$ programs, as

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

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

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

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

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

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

Figure 4: Type-Directed GADT to $\mathrm{F_C}$ Translation (interesting cases)

GADT programs only make use of syntactic (a.k.a. Herbrand) type equality, and so, require no type functions at all.

THEOREM 3 (GADT Consistency). *If $dom(\Gamma)$ contains no type variables or coercion constants, and $\Gamma \vdash_{\text{CO}} \gamma : \sigma_1 \doteq \sigma_2$, 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 3. Hence, all GADT $\mathrm{F_C}$ programs are sound. From the erasure soundness (cf. Appendix A), we can immediately conclude that the semantics of GADT programs remains unchanged (where e° is e after type erasure).

LEMMA 2. *Let $\emptyset; \emptyset \vdash_{GADT} e : t \rightsquigarrow e'$. 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 $\mathrm{F_C}$ 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 $\mathrm{F_C}$ is almost embarrassingly simple, especially given the translation of GADTs to $\mathrm{F_C}$ from §4. In the following, we outline the additions required to the standard translation of type classes to System F [15] to support associated types.

5.1 Translating expressions

To translate expressions, we need to add three rules to the standard system of [15], namely Rules (Eq), (C-Intro), and (C-Elim) from Figure 4 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 $C \Vdash_D D\ \tau \rightsquigarrow \nu$. 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:

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

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 $\mathrm{F_C}$, and each definition of an associated type in an instance turns into an equality **axiom** in $\mathrm{F_C}$. We have seen examples of this in §2.2.

5.4 Observations

In the translation of associated types, it becomes clear why $\mathrm{F_C}$ 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 $\mathrm{F_C}$.

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] $C\ :\ (\forall\overline{a:\star}.S\ \sigma_1) \doteq (\forall\overline{a:\star}.\sigma_2)$. where (a) σ_1 does not refer to any type function, (b) $\mathit{fv}(\sigma_1) = \bar{a}$, and (c) $\mathit{fv}(\sigma_2) \subseteq \bar{a}$. 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 4 (Canonical Normal Forms). *Let Γ be well-formed, terminating and confluent. Then, $\Gamma \vdash_{\mathrm{co}} \gamma : \sigma_1 \doteq \sigma_2$ 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 2 (AT Consistency). *If Γ contains only rewrite axioms that together form a terminating and confluent rewrite system, then Γ is consistent.*

For example, assume $\Gamma \vdash_{\mathrm{co}} \gamma : T_1\ \overline{\sigma_1} \doteq T_2\ \overline{\sigma_2}$. 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 $\Gamma = \{c_1 : S_1\ [Int] \doteq S_2, c_2 : (\forall a:\star.S_1\ [a]) \doteq (\forall a:\star.[S_1\ a])\}$. We find that $\Gamma \vdash_{\mathrm{co}} \gamma : S_1\ [Int] \doteq S_2$. 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.

6. Related Work

System F with GADTs. Xi et al.[39] 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 [31]. This is the approach that GHC initially took. $\mathrm{F_C}$ is the result of a search for an alternative.

Encoding GADTs in plain System F and $\mathbf{F}_\omega$. There are several previous works [3, 9, 26, 38, 7, 36] 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 B.

An encoding of a restricted subset of GADT programs in plain System F_ω can be found in [29], but this encoding only works for limited patterns of recursion.

Coercion-based subtyping. Mitchell [25] 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 [19]. Subsequently, Mitchell [24] also studied coercions that are operationally identities to model type refinement for type inference in systems that go beyond Hindley/Milner.

Much closer to $\mathrm{F_C}$ 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 $\mathrm{F_C}$, their coercions model a subsumption relationship, not equalities, and their coercions are values, not types. Nevertheless, they introduce coercion combinators, as we do, but they don't consider decomposition, which is crucial to translate GADTs. Moreover, 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 [25, 24] 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 [16], or generally the evidence in intuitionistic type theory [22]. This connection indicates several directions for extending the presented system in the direction of more powerful dependently typed languages, such as Epigram [23].

Shao et al. [32] use equalities for type conversion in a lambda calculus with a type language based on the calculus of inductive constructions to denote certified binaries. Interestingly, already GADT equalities are useful to encode safety properties, as Sheard [33] demonstrated.

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 [13]. Our use of equalities to represent programs that involve functional dependencies and existential types seems related.

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, functional dependencies, and closed type functions. An interesting avenue for future work is to find good source language features to expose more of the power of $\mathrm{F_C}$ to programmers.

Acknowledgements. We thank James Cheney, Roman Leshchinskiy, Conor McBride, Benjamin Pierce, and François Pottier for

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

their helpful comments on previous versions of this paper. We are grateful to Lennart Augustsson for a discussion on encoding associated types in System F and thank Andreas Abel for pointing out the connection to [29].

References

[1] M. Abadi, L. Cardelli, and P.-L. Curien. Formal parametric polymorphism. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 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 *ICFP '05: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, 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 *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 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 *POPL'03: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 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 *ICFP'00: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 68–81, New York, NY, USA, 2000. ACM Press.

[12] 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.

[13] D. Dreyer, K. Crary, and R. Harper. A type system for higher-order modules. In *POPL'03: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 236–249, New York, NY, USA, 2003. ACM Press.

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

[15] 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.

[16] R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. In *Proceedings 2nd Annual IEEE Symp. on Logic in Computer Science, LICS'87*, pages 194–204. IEEE Computer Society Press, 1987.

[17] 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.

[18] 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.

[19] R. Kießling and Z. Luo. Coercions in hindley-milner systems. In *Types for Proofs and Programs: Third International Workshop, TYPES 2003, Revised Selected Papers*, number 3085 in LNCS, pages 259–275, 2004.

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

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

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

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

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

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

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

[27] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. `http://research.microsoft.com/~simonpj/papers/gadt/`, Nov. 2005.

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

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

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

[31] 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.

[32] 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.

[33] 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.

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

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

[36] 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.

[37] 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.

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

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

A. Type and coercion erasure

In this section we give the details of our type erasure theorem. Following Pierce [28, 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.

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

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

THEOREM 5. *Given $\Gamma \vdash_e e_1 : \sigma$, (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. 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. □

COROLLARY 3 (Erasure soundness). *For an well-typed System $\mathrm{F_C}$ term e_1, we have $e_1 \longrightarrow^* e_2$ iff ${e_1}^\circ \longrightarrow^* {e_2}^\circ$.*

B. 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 $a \doteq b$ 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, 26, 38], 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 [36]. 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 [36]. 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
```

First, we convince ourselves that the above program is well-typed. The pattern `I x` in combination with the type annotation implies that `Foo a` $=$ `Foo Int`. By decomposition, we conclude that `a` $=$ `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 [15].

```
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 [15], 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.

C. Complexity of Type Checking

Previous calculi for GADTs, such as $\lambda_{2,G\mu}$ [39] and MLGX [31], 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 $\mathrm{F_{C_i}}$ and show that type checking for $\mathrm{F_{C_i}}$ is undecidable. More precisely, we show that reconstructing explicit coercion terms, which amount to proofs justifying coercions, is undecidable for $\mathrm{F_{C_i}}$.

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

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

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 (TApp) and (Cast) by those in Figure 5; for all other rules, the translation is the identity. The modified Rules (Cast$_i$) and (TApp$_{\mathrm{Co}}$) use the judgement $\Gamma \vdash_{\mathrm{Co}} \gamma : \sigma \doteq \tau$ to re-compute γ. As we will see next, computing γ from a kind $\sigma \doteq \tau$ 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 $\Gamma \vdash_e e \rightsquigarrow e' : \sigma$ 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 [30]. 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:

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

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

$$(\text{TApp}_{\text{TY}})\quad \frac{\Gamma \vdash_e e : \forall a{:}\kappa.\sigma \quad \Gamma \vdash_k \kappa : \text{TY} \quad \Gamma \vdash_{\text{TY}} \tau : \kappa}{\Gamma \vdash_e (e\,@\,\tau) \rightsquigarrow (e\,@\,\tau) : \sigma[\tau/a]} \qquad (\text{TApp}_{\text{CO}})\quad \frac{\Gamma \vdash_e e : \forall a{:}(\tau \doteq \upsilon).\sigma \quad \Gamma \vdash_k (\tau \doteq \upsilon) : \text{CO} \quad \Gamma \vdash_{\text{CO}} \varphi : \tau \doteq \upsilon}{\Gamma \vdash_e (e\,@\{\tau \doteq \upsilon\}) \rightsquigarrow (e\,@\,\varphi) : \sigma[\varphi/a]}$$

Figure 5: Modified typing rules for System $\mathrm{F}_{\mathrm{C}_i}$

The data type $Term$ enables us to construct any (ground) $\mathcal{F}$-term by reflection from the structurally identical 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 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 $\mathrm{F}_{\mathrm{C}_i}$ type checking problem by typing the cast expression $\widehat{\sigma} \blacktriangleright \{\sigma \doteq \tau\}$ in the context of the above F_{C} declarations corresponding to $\mathcal{F}$ and E. The undecidability of the word problem implies the undecidability of $\mathrm{F}_{\mathrm{C}_i}$ typing, or more precisely, that the judgement $\Gamma \vdash_{\text{CO}} \gamma : \sigma \doteq \tau$ in the premise of $\mathrm{F}_{\mathrm{C}_i}$'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 $\mathrm{F}_{\mathrm{C}_i}$ 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 [34], this is not a viable approach.