Injective Type Families for Haskell

Jan Stolarek
Politechnika Łódzka, Poland
jan.stolarek@p.lodz.pl

Simon Peyton Jones
Microsoft Research Cambridge, UK
simonpj@microsoft.com

Richard A. Eisenberg
University of Pennsylvania, PA, USA
eir@cis.upenn.edu

Abstract

Haskell, as implemented by the Glasgow Haskell Compiler (GHC), allows expressive type-level programming. The most popular type-level programming extension is *TypeFamilies*, which allows users to write functions on types. Yet, using type functions can cripple type inference in certain situations. In particular, lack of injectivity in type functions means that GHC can never infer an instantiation of a type variable appearing only under type functions.

In this paper, we describe a small modification to GHC that allows type functions to be annotated as injective. GHC naturally must check validity of the injectivity annotations. The algorithm to do so is surprisingly subtle. We prove soundness for a simplification of our algorithm, and state and prove a completeness property, though the algorithm is not fully complete.

As much of our reasoning surrounds functions defined by a simple pattern-matching structure, we believe our results extend beyond just Haskell. We have implemented our solution on a branch of GHC and plan to make it available to regular users with the next stable release of the compiler.

Categories and Subject Descriptors F.3.3 [*Logics And Meanings Of Programs*]: Studies of Program Constructs – Type structure; D.3.1 [*Programming Languages*]: Formal Definitions and Theory – Semantics; D.3.2 [*Programming Languages*]: Language Classifications – Haskell

Keywords Haskell; type-level programming; type families; functional dependencies; injectivity

1. Introduction

The Glasgow Haskell Compiler (GHC) offers many language extensions that facilitate type-level programming. These extensions include generalized algebraic data types (GADTs) (Cheney and Hinze 2003; Peyton Jones et al. 2006), datatype promotion with kind polymorphism (Yorgey et al. 2012), and functional dependencies (Jones 2000). But the most widespread[1] extension for type-level programming is for *type families*, which allow users to define type-level functions (Chakravarty et al. 2005a,b; Eisenberg et al. 2014) run by the type checker during compilation. Combined with other features, they allow expressiveness comparable to that of languages with dependent types (Lindley and McBride 2013).

However, type families as implemented in GHC have a serious deficiency: they cannot be declared to be *injective*. Injectivity is very important for type inference: without injectivity, some useful functions become effectively unusable, or unbearably clumsy. *Functional dependencies*, which have been part of GHC for many years, are arguably less convenient (Section 7), but they certainly can be used to express injectivity. That leaves programmers with an awkward choice between the two features.

In this paper we bridge the gap, by allowing programmers to declare their type functions injective, while the compiler checks that their claims are sound. Although this seems straightforward, it turned out to be much more subtle than we expected. Our main contribution is to identify and solve these subtleties. Although our concrete focus is on Haskell, our findings apply to any language that defines functions via pattern matching and allows to run them during compilation. Specifically:

- We introduce a backwards-compatible extension to type families, which allows users to annotate their type family declarations with information about injectivity (Section 3).
- We give a series of examples that illustrate the subtleties of checking injectivity (Section 4.1).
- We present a *compositional* algorithm for checking whether a given type family (which may be open or closed) is injective (Section 4.2), and prove it sound (Section 4.3). We show that a compositional algorithm cannot be complete, but nevertheless give a completeness proof for a sub-case where it holds (Section 4.4).
- We explain how injectivity information can be exploited by the type inference algorithm, including elaboration into GHC's statically typed intermediate language, System FC (Section 5).
- We describe how to make the injectivity framework work in the presence of kind polymorphism (Section 6).
- We provide an implementation of our solution in a development branch of GHC. We expect it to become available to regular users with the next stable release.

Our work is particularly closely related to functional dependencies, as we discuss in Section 7, leaving other related work for Section 8.

An extended version of the paper is available online, with proofs of the theorems (Stolarek et al. 2015).

2. Why Injective Type Families Matter

We begin with a brief introduction to type families, followed by motivating examples, inspired by real bug reports, that illustrate why injectivity is important.

[1] Appendix A gives data and describes our methodology for obtaining them.

Haskell'15, September 3-4, 2015, Vancouver, BC, Canada

http://dx.doi.org/10.1145/2804302.2804314

$$
\begin{array}{rcll}
\sigma, \tau & ::= & \alpha & \text{Type variable} \\
& | & H & \text{Type constructor} \\
& | & \tau_1\ \tau_2 & \text{Application} \\
& | & F\ \overline{\tau} & \text{Saturated type-function application}
\end{array}
$$

The forms $(\tau_1\ \tau_2)$ and $(F\ \overline{\tau})$ are syntactically distinct, and do not overlap despite the similarity of their concrete syntax.

Figure 1. Syntax of types.

2.1 Type Families in Haskell

Haskell (or, more precisely, GHC), supports two kinds of type family: *open* and *closed*[2]. An open type family (Chakravarty et al. 2005a,b) is specified by a **type family** declaration that gives its arity, its kind (optionally), and zero or more **type instance** declarations that give its behaviour. For example:

$$
\begin{array}{l}
\textbf{type family}\ F\ a \\
\textbf{type instance}\ F\ Int = Bool \\
\textbf{type instance}\ F\ [a] = a \to a
\end{array}
$$

The type-instance equations may be scattered across different modules and are unordered; if they overlap they must be *compatible*. We say that two overlapping type family equations are compatible when any application matching both of these equations reduces, in one step, to the same result with any of these equations.

A closed type family (Eisenberg et al. 2014) is declared with all its equations in one place. The equations may overlap, and are matched top-to-bottom. For example:

$$
\begin{array}{l}
\textbf{type family}\ G\ a\ \textbf{where} \\
\quad G\ Int = Bool \\
\quad G\ a\ \ = Char
\end{array}
$$

In both open and closed case the family is defined by zero[3] or more *equations*, each of form $F\ \overline{\tau} = \sigma$, where the *left hand side* (LHS) of the equation is $F\ \overline{\tau}$, the *right hand side* (RHS) is σ, and:

- Every left hand side has the same number of argument types $\overline{\tau}$; this number is the *arity* of the family.
- Every type variable mentioned on the right must be bound on the left: $\mathit{ftv}(\overline{\tau}) \supseteq \mathit{ftv}(\sigma)$.
- The types $\overline{\tau}$ and σ must be monotypes; they contain no for-all quantifiers.
- In addition, the types $\overline{\tau}$ on the left hand side must be type-function-free.

For the purposes of Sections 3–5 we restrict our attention to kind-monomorphic type functions. The generalization to polymorphic kinds is straightforward – see Section 6.

Type functions may only appear *saturated* in types. That is, if F has arity 2, it must always appear applied to two arguments. Figure 1 gives the syntax of (mono-)types.

Finally, type functions may be *partial*. For example, referring to F above, the type $(F\ Char)$ matches no equation. Nevertheless, the type $(F\ Char)$ is not immediately an error in Haskell, as one might expect if F were a term-level function. Instead, $(F\ Char)$ is an uninhabited type (except by $\bot$), equal to no other type. This design decision is forced by the notion of open type families, since new types are declared all the time in Haskell, and we could not possibly insist on giving them a **type instance** declaration for every (usually-irrelevant) type family.

2.2 The Need for Injectivity

Our work on injective type families began in response to user requests[4] for this feature. Here is a boiled-down version of one suggested use case:

$$
\begin{array}{l}
\textbf{class}\ Manifold\ a\ \textbf{where} \\
\quad \textbf{type}\ Base\ a \\
\quad project\ \ \ :: a \to Base\ a \\
\quad unproject :: Base\ a \to a \\
id :: forall\ a.\ (Manifold\ a) \Rightarrow Base\ a \to Base\ a \\
id = project\ \circ\ unproject
\end{array}
$$

The id function composes unprojection and projection, effectively normalising the representation of base vectors in a manifold. However, GHC rejects type signature of id as ambiguous since the type variable a appears only under type family applications and in the set of constraints. This means that, at the call site of id, the compiler is unable to determine how a should be instantiated. For example, consider a call $(id\ \ vec)$, where $vec :: [Double]$. What type should instantiate a in the call? Clearly, we should pick such a that $Base\ a = [Double]$. But if $Base$ were not injective, there could be many valid choices for a, each with its own instance for $Manifold$. The choice for a thus affects runtime behaviour – it absolutely must not be left to the whim of a compiler. Since there is no unique choice, GHC refrains from guessing, and instead reports a as an ambiguous type variable.

A similar problem arises in the `vector` library, which provides efficient implementation of integer-indexed arrays[5]. The `vector` library defines an open type family, $Mutable$, that assigns a mutable counterpart for every immutable vector type:

$$
\textbf{type family}\ Mutable\ v
$$

For example, if $ByteString$ is an immutable vector of bytes, and $MByteString$ is the mutable variant, then we can express the connection by writing[6]:

$$
\textbf{type instance}\ Mutable\ ByteString = MByteString
$$

The library also provides two functions over vectors:

$$
\begin{array}{l}
freeze\ \ \ :: Mutable\ v \to IO\ v \\
convert :: (Vector\ v, Vector\ w) \Rightarrow v \to w
\end{array}
$$

$freeze$ takes a mutable vector and turns it into an immutable one; $convert$ converts one kind of vector into another. But now suppose the programmer writes this:

$$
\begin{array}{l}
ftc :: (Vector\ v, Vector\ w) \Rightarrow Mutable\ v \to IO\ w \\
ftc\ mv = \textbf{do}\ \{ v \leftarrow freeze\ mv; return\ (convert\ v) \}
\end{array}
$$

Again GHC complains that the type of ftc is ambiguous: in a call $(ftc\ \ vec)$, where $vec :: MByteString$, it is not clear how to instantiate v. GHC correctly reports v as an ambiguous type variable.

Current Solution: Proxies To resolve such ambiguities the programmer must give guidance to the type inference engine. A standard idiom in cases like these is to use *proxy arguments*. For example we could rewrite ftc like this:

[2] *Associated types* (Chakravarty et al. 2005a) are essentially syntactic sugar for open type families. Everything we say in this paper works equally for associated types, both in theory and in the implementation. So we do not mention associated types further, apart from a short discussion in Section 3.3.

[3] Empty closed type families are implemented in the development version of GHC and will be available to regular users with the next stable release.

[4] `https://ghc.haskell.org/trac/ghc/ticket/6018`

[5] `https://github.com/haskell/vector/issues/34`

[6] In the real library, the argument to $Mutable$ is the type *constructor* for a vector; but that genererality complicates the example and obscures the main point, so we use a simpler, monomorphic version here.

```
data Proxy a
ftc' :: (Vector v, Vector w)
     ⇒ Proxy v   -- NB: extra argument here
     → Mutable v → IO w
ftc' _ mv = do {v ← freeze v; return (convert v)}
```

Instead of the problematic call $(ftc\ vec)$ where $vec :: MByteString$, the programmer must supply an explicit proxy argument, thus:

$$ftc'\ (\bot :: Proxy\ ByteString)\ vec$$

The value of the proxy argument is $\bot$; its only role is to tell the type inference engine to instantiate the type variable v to $ByteString$.

This works, but it is absurdly clumsy, forcing the programmer to supply redundant arguments. Why redundant? Because, *in the programmer's mind*, if we know, say, that $Mutable\ a$ is $MByteString$ then we know that a must be $ByteString$. That is, every immutable vector type has its own unique mutable counterpart; more precisely, $Mutable$ is an injective function. We simply need a way for the library author to express that property to the compiler.

Our Solution: Injective Type Families In this paper we therefore allow programmers to declare a type function to be injective, using an *injectivity annotation*, thus:

```
class Manifold a where
  type Base a = r | r → a
  ...
type family Mutable v = r | r → v
```

The user names the result of the type family as r and, using syntax inspired by functional dependencies, declares that the result r determines the argument. GHC then verifies that the injectivity annotation provided by the user holds for every type family instance. During type inference, GHC can exploit injectivity to resolve type ambiguity. This solves the problems with $Manifold$ and the vector library with one line apiece – no other changes are required.

3. Injective Type Families

Next we describe our proposed extension, from the programmer's point of view.

3.1 Injectivity of Type Families

In the rest of this paper we depend on the following definition of injectivity for type families, whether open or closed:

Definition 1 (Injectivity)**.** *A type family F is n-injective (i.e. injective in its n'th argument) iff* $\forall \overline{\sigma}, \overline{\tau}\colon F\ \overline{\sigma} \sim F\ \overline{\tau} \implies \sigma_n \sim \tau_n$

Here $\sigma \sim \tau$ means that we have a proof of equality of types σ and τ. So the definition simply says that if we have a proof that $F\,\overline{\sigma}$ is equal to $F\,\overline{\tau}$, then we have a proof that σ_n and τ_n are equal. Moreover, if we know that $F\,\overline{\tau} \sim \tau'$, and τ' is known, we can discover values of injective arguments $\overline{\tau_n}$ by looking at the defining equation of F that has right-hand side matching τ'. Section 5 provides the details.

3.2 Annotating a Type Family with Injectivity Information

Injectivity is a subtle property and *inferring* it is not necessarily possible or desirable (see Section 3.4), so we therefore ask the user to *declare* it. The compiler should check that the declared injectivity of a type family is sound.

What syntax should we use for such injectivity annotations? We wanted to combine full backwards compatibility when injectivity is not used, and future extensibility (Section 7 discusses the latter). Definition 1 admits injectivity in only some of the arguments and so we have to be able to declare that a function is injective in its second argument (say) but not its first.

To achieve this, we simply allow the programmer to name the result type and, using a notation borrowed from functional dependencies (Jones 2000), say which arguments are determined by the result. For example:

```
type family F a b c = r | r → a c
```

The "$= r$" part names the result type, while the "$r \to a\ c$" – termed the *injectivity condition* – says that the result r determines arguments a and c, but not b. The result variable may be annotated with a kind, and the injectivity part is optional. So all of the following are legal definitions:

```
type family F a b c
type family F a b c = r
type family F a b c = (r :: * → *) | r → a
type family F a b c = r | r → a c
```

Examples above use *open* type families but the syntax also extends to *closed* type families, where the injectivity annotation precedes the **where** keyword.

3.3 Associated Types

A minor syntactic collision occurs for *associated types*:

```
class C a b where
  type F a b
  type F a b = b
```

The second line beginning "**type** $F\ a\ b$" is taken as the default instance for the associated type (to be used in instances of C in which F is not explicitly defined). Note that the **family** and **instance** keywords can be omitted for associated types and that the default *instance* **type** $F\ a\ b = b$ looks suspiciously like a type *family* with a named result type. To avoid this ambiguity, you can only name the result type with associated types if you also give an injectivity annotation, thus:

```
class C a b where
  type F a b = r | r → b
  type F a b = b
```

As explained in Section 4, GHC must check instances of injective type families to make sure they adhere to the injectivity criteria. For associated type defaults, the checks are made only with concrete instances (that is, when the default is actually used in a class instance), not when processing the default declaration. This choice of behaviour is strictly more permissive than checking defaults at the class declaration site.

3.4 Why not Infer Injectivity?

One can wonder why we require explicit annotations rather then inferring injectivity.

For open type families, inferring injectivity is generally impossible, as the equations are spread across modules and can be added at any time. Inferring injectivity based only on those equations in the declaring module would lead to unexpected behaviour that would arise when a programmer moves instances among modules.

Inferring injectivity on *closed* type families, however, is theoretically possible, but we feel it is the wrong design decision, as it could lead to unexpected behaviour during code refactoring. An injectivity declaration states that the injectivity property of a type family is required for the program to compile. If injectivity were inferred, the user might be unaware that she is relying on injectivity. Say our programmer has an inferred-injective type family F. She then adds a new equation to the definition of F that breaks the injectivity property. She could easily be surprised that, suddenly, she has compilation errors in distant modules, if those modules (perhaps

unwittingly) relied on the injectivity of F. Even worse, the newly-erroneous modules might be in a completely different package. With the requirement of an explicit annotation, GHC reports an error at the offending type family equation. To keep matters simple we restrict ourselves to explicitly-declared injectivity.

4. Verifying Injectivity Annotations

Before the compiler can exploit injectivity (Section 5), it must first check that the user's declaration of injectivity is in fact justified. In this section we give a sound, compositional algorithm for checking injectivity, for both open and closed type functions.

We want our algorithm to be *compositional* or *modular*: that is, we can verify injectivity of function F by examining only the equations for F, perhaps making use of the declared injectivity of other functions. In contrast a non-compositional algorithm would require a global analysis of all functions simultaneously; that is, a compositional algorithm is necessarily incomplete. A non-compositional algorithm would be able to prove more functions injective (Section 4.4), but at the expense of complexity and predictability. A contribution of this paper is to articulate a compositional algorithm, and to explain exactly what limitations it causes.

Soundness means that if the algorithm declares a function injective, then it really is; this is essential (Section 4.3). *Completeness* would mean that if the function really is injective, then the algorithm will prove it so. Sadly, as we discuss in Section 4.4, completeness is incompatible with compositionality. Nevertheless we can prove completeness for a sub-case.

4.1 Three Awkward Cases

Checking injectivity is more subtle than it might appear. Here are three difficulties, presented in order of increasing obscurity.

Awkward Case 1: Injectivity is not Compositional First consider this example:

$$\begin{array}{l}\textsf{type family } F1\ a = r \mid r \to a\\ \textsf{type instance } F1\ [a] \quad\ = G\ a\\ \textsf{type instance } F1\ (Maybe\ a) = H\ a\end{array}$$

Is $F1$ injective, as claimed? Even if G and H are injective, there is no guarantee that $F1$ is, at least not without inspecting the definitions of G and H. For example, suppose we have:

$$\begin{array}{l}\textsf{type instance } G\ Int\ \ = Bool\\ \textsf{type instance } H\ Bool = Bool\end{array}$$

So both G and H are injective. But $F1$ is clearly not injective; for example $F1\ [Int] \sim G\ Int \sim Bool \sim H\ Bool \sim F1\ (Maybe\ Bool)$. Thus, injectivity is not a compositional property.

However, it is over-conservative to reject *any* type function with type functions in its right-hand side. For example, suppose G and H are injective, and consider $F2$ defined thus:

$$\begin{array}{l}\textsf{type family } F2\ a = r \mid r \to a\\ \textsf{type instance } F2\ [a] \quad\ = [G\ a]\\ \textsf{type instance } F2\ (Maybe\ a) = H\ a \to Int\end{array}$$

Since a list cannot possibly match a function arrow, an equality $(F2\,\sigma \sim F2\,\tau)$ can only hold by using the same equation twice; and in both cases individually the RHS determines the LHS because of the injectivity of G and H. But what about these cases?

$$\begin{array}{l}\textsf{type family } F3\ a = r \mid r \to a\\ \textsf{type instance } F3\ [a] \quad\ = Maybe\ (G\ a)\\ \textsf{type instance } F3\ (Maybe\ a) = Maybe\ (H\ a)\\ \textsf{type family } F4\ a = r \mid r \to a\\ \textsf{type instance } F4\ [a] \quad\ = (G\ a, a, a, a)\\ \textsf{type instance } F4\ (Maybe\ a) = (H\ a, a, Int, Bool)\end{array}$$

$F3$ is not injective, for the same reason as $F1$. But $F4$ *is* injective, because, despite calls to two different type families appearing as the first component of a tuple, the other parts of the RHSs ensure that they cannot unify.

Awkward Case 2: The Right Hand Side Cannot Be a Bare Variable or Type Family The second awkward case is illustrated by this example:

$$\begin{array}{l}\textsf{type family } W1\ a = r \mid r \to a\\ \textsf{type instance } W1\ [a] = a\end{array}$$

To a mathematician this function certainly *looks* injective. But, surprisingly, it does not satisfy Definition 1! Here is a counter-example. Clearly we do have a proof of $(W1\ \ [W1\ \ Int] \sim W1\ \ Int)$, simply by instantiating the **type instance** with $[a \mapsto W1\ \ Int]$. But if $W1$ was injective in the sense of Definition 1, we could derive a proof of $[\,W1\ \ Int] \sim Int$, and that is plainly false! Similarly:

$$\begin{array}{l}\textsf{type family } W2\ a = r \mid r \to a\\ \textsf{type instance } W2\ [a] = W2\ a\end{array}$$

Again $W2$ looks injective. But we can prove $W2\ [Int] \sim W2\ Int$, simply by instantiating the type instance; then by Definition 1, we could then conclude $[Int] \sim Int$, which is plainly false. So neither $W1$ nor $W2$ are injective, according to our definition. Note that the partiality of $W1$ and $W2$ is critical for the failure case to occur.

Awkward Case 3: Infinite Types Our last tricky case is exemplified by Z here:

$$\begin{array}{l}\textsf{type family } Z\ a = r \mid r \to a\\ \textsf{type instance } Z\ [a] \quad\ = (a, a)\\ \textsf{type instance } Z\ (Maybe\ b) = (b, [b])\end{array}$$

Quick: is Z injective? Are there any types s and t for which $Z\ \ [t] \sim Z\ \ (Maybe\ \ s)$? Well, by reducing both sides of this equality that would require $(t, t) \sim (s, [s])$. Is that possible? You might think not – after all, the two types do not unify. But consider G, below:

$$\begin{array}{l}\textsf{type family } G\ a\\ \textsf{type instance } G\ a = [G\ a]\end{array}$$

(Whether or not G is injective is irrelevant.) Now choose $t = s = G\ \ Int$. We have $Z\ \ [G\ \ Int] \sim (G\ \ Int, G\ \ Int) \sim (G\ \ Int, [G\ \ Int]) \sim Z\ \ (Maybe\ \ (G\ \ Int))$. Now use Definition 1 on the first and last of this chain of equalities, to deduce $[G\ \ Int] \sim Maybe\ \ (G\ \ Int)$, which is unsound. Indeed $(t, t) \sim (s, [s])$ holds! And so Z is not injective, according to Definition 1.

One reasonable way to fix this is to guarantee that all type-level functions are terminating, so that there are no infinite types like $G\ Int$. This is GHC's default behaviour (Chakravarty et al. 2005a), but it comes at the cost of restricting the form of type-function definitions. (After all, termination is an undecidable property.) GHC therefore offers the *UndecidableInstances* extension, which lifts the restrictions that guarantee termination, for both type classes and type functions. If a type function diverges the type checker may loop, and that seems fair enough.

Our concern with combining injectivity with *UndecidableInstances* is that the type checker might terminate, *but generate an unsound program*, and that is unacceptable. As long as GHC accepts potentially non-terminating type families, the possibility of such a disaster is real, and we must guard against it.

4.2 The Injectivity Check

Equipped with these intuitions, we can give the following injectivity-check algorithm:

Definition 2 (Injectivity check)**.** *A type family F is n-injective iff*

1. *For every equation $F\ \overline{\sigma} = \tau$:*
   (a) τ is not a type family application, and
   (b) if $\tau = a_i$ (for some type variable a_i), then $\overline{\sigma} = \overline{a}$.
2. *Every pair of equations $F\ \overline{\sigma}_i = \tau_i$ and $F\ \overline{\sigma}_j = \tau_j$ (including $i = j$) is* pairwise-n-injective*.*

Clause 2 compares equations pairwise (Section 4.5 discusses separate compilation). Here is the intuition, which we will make precise in subsequent sections:

Definition 3 (Intuitive pairwise check)**.** *Two equations are pairwise-n-injective if, when the RHSs are the same, then the n'th argument on the left hand sides are also the same.*

Clause 1 deals with Awkward Case 2, by rejecting equations whose RHS is a bare type variable or function call. This restriction is barely noticeable in practice, because any equation rejected by Clause 1 would *also* be rejected by Clause 2, if there was more than one equation. That leaves only single-equation families, such as

type instance $F\ a = G\ a$

which might as well be done with a type synonym. The sole exception are equations of the form $F\ a\ b = a$, a useful fallthrough case for a closed type family. We allow this as a special case; hence 1b.

Notice that Clause 1 permits a RHS that is *headed by* a type variable or type function application; e.g. $F\ (T\ a\ b) = a\ b$ or $F\ [a] = G\ a\ Int$, where G has an arity of 1.

4.2.1 Unifying RHSs

In the intuitive injectivity check above, we check if two RHSs are the same. However, type families are, of course, parameterized over variables, so the "sameness" check must really mean unification. For example:

type family $G1\ a = r \mid r \to a$
type instance $G1\ [a] = [a]$
type instance $G1\ (Maybe\ b) = [(b, b)]$

It would be terribly wrong to conclude that $G1$ is injective, just because a and (b, b) are not syntactically identical.

Unifying the RHSs will, upon success, yield a substitution. We want to apply that substitution to the LHSs, to see if they become syntactically identical. For example, consider:

type family $G2\ a\ b = r \mid r \to a\ b$
type instance $G2\ a\ \ Bool = (a, a)$
type instance $G2\ Bool\ b\ \ = (b, Bool)$

Unifying the RHSs yields a most-general substitution that sets both a and b to $Bool$. Under this substitution, the LHSs are the same, and thus $G2$ is injective.

We must be careful about variable names however. Consider $G3$:

type family $G3\ a\ b = r \mid r \to b$
type instance $G3\ a\ Int\ \ = (a, Int)$
type instance $G3\ a\ Bool = (Bool, a)$

This function is not injective: both $G3\ Bool\ Int$ and $G3\ Int\ Bool$ reduce to $(Bool, Int)$. But the RHSs, as stated, do not unify: the unification algorithm will try to set a to both $Bool$ and Int. The solution is simple: freshen type variables, so that the sets of variables in the equations being compared are disjoint. In this example, if we freshen the a in the second equation to b, we get a unifying substitution $[a \mapsto Bool, b \mapsto Int]$, and since the LHSs do not coincide under that substitution, we conclude that $G3$ is not injective.

$$
\begin{array}{rll}
U(a,\tau)\ \theta = U(\theta(a),\tau)\ \theta & a \in \mathsf{dom}(\theta) & (1)\\
U(a,\tau)\ \theta = \textbf{Just}\ \theta & a \in ftv(\theta(\tau)) & (2)\\
U(a,\tau)\ \theta = \textbf{Just}\ ([a \mapsto \theta(\tau)] \circ \theta) & a \notin ftv(\theta(\tau)) & (3)\\
U(\tau,a)\ \theta = U(a,\tau)\ \theta & & (4)\\
U(\sigma_1\ \sigma_2, \tau_1\ \tau_2)\ \theta = U(\sigma_1,\tau_1)\ \theta \gg\!= U(\sigma_2,\tau_2) & & (5)\\
U(H,H)\ \theta = \textbf{Just}\ \theta & & (6)\\
U(F\ \overline{\sigma}, F\ \overline{\tau})\ \theta = U(\sigma_i,\tau_i)\ \theta & F \text{ is } i\text{-injective} & (7)\\
\gg\!= \quad \ldots & \text{...etc...} & \\
\gg\!= \quad U(\sigma_j,\tau_j) & F \text{ is } j\text{-injective} & \\
U(F\ \overline{\sigma},\tau)\ \theta = \textbf{Just}\ \theta & & (8)\\
U(\tau,F\ \overline{\sigma})\ \theta = \textbf{Just}\ \theta & & (9)\\
U(\sigma,\tau)\ \theta = \textbf{Nothing} & & (10)
\end{array}
$$

$$
\begin{array}{rl}
\textbf{Just}\ \theta \gg\!= k & = k\ \theta\\
\textbf{Nothing} \gg\!= k & = \textbf{Nothing}
\end{array}
$$

Note that equations (5) and (7) do not overlap, despite appearing to do so. See the note in Figure 1.

Figure 2. Pre-unification algorithm U.

Conveniently, freshening variables and unifying allows us to cover one other corner case, exemplified in $G4$:

type family $G4\ a\ b = r \mid r \to a\ b$
type instance $G4\ a\ b = [a]$

The type family $G4$ is not injective in its second argument, and we can see that by comparing the equation *against itself*; that is, when we say "every pair of equations" in Definition 3 we include the pair of an equation with itself. When comparing $G4$'s single equation with itself, variable freshening means that we effectively compare:

type instance $G4\ a1\ b1 = [a1]$
type instance $G4\ a2\ b2 = [a2]$

The unifying substitution can be $[a1 \mapsto a2]$. Applying this to the LHSs still yields a conflict $b1 \neq b2$, and $G4$ is (rightly) discovered to be non-injective. Summing this all together, we can refine our intuitive pairwise check as follows:

Definition 4 (Unsound pairwise check)**.** *Two equations $F\ \overline{\sigma}_i = \tau_i$ and $F\ \overline{\sigma}_j = \tau_j$, whose variables are disjoint*[7]*, are pairwise-n-injective iff either*

1. *Their RHSs τ_i and τ_j fail to unify, or*
2. *Their RHSs τ_i and τ_j unify with substitution θ, and $\theta(\sigma_{i\,n}) = \theta(\sigma_{j\,n})$.*

Alas, as we saw in Awkward Case 1 (Section 4.1), if the RHS of a type instance can mention a type family, this test is unsound. We explain and tackle that problem next.

4.2.2 Type Families on the RHS

If the RHS of a type instance can mention a type family, classical unification is not enough. Consider this example:

type family $G5\ a = r \mid r \to a$
type instance $G5\ [a] = [G\ a]$
type instance $G5\ Int = [Bool]$

Here, G is some other type family, known to be injective. When comparing these equations, the RHSs do not unify under the classical definition of unification (i.e. there is no unifying substitution).

[7] We can always make them disjoint by α-conversion.

Therefore, under Definition 4, $G5$ would be accepted as injective. However, this is wrong: we might have $G\ Int = Bool$, in which case $G5$ is plainly not injective.

To fix this problem, we need a variant of the unification algorithm that *treats a type family application as potentially unifiable with any other type*. Algorithm $U(\sigma, \tau)\ \theta$ is defined in Figure 2. It takes types σ and τ and a substitution θ, and returns one of two possible outcomes: **Nothing**, or **Just** ϕ, where ϕ extends θ. We say that ϕ *extends* θ iff there is a (possibly empty) θ' such that $\phi = \theta' \circ \theta$.

The definition is similar to that of classical unification except:

- Equations (8) and (9) deal with the case of a type-function application; it immediately succeeds without extending the substitution.
- Equation (7) allows U to recurse into *the injective arguments of* a type-function application.
- Equation (2) would fail in classical unification (an "occurs check"); U succeeds immediately, but without extending the substitution. We discuss this case in Section 4.2.3.

We often abbreviate $U(\sigma, \tau)\ \varnothing$ as just $U(\sigma, \tau)$, where $\varnothing$ is the empty substitution.

Algorithm U has the following two properties, which may be proved in a similar manner to the proof of correctness of Robinson's unification algorithm.

Property 5 (No false negatives)**.** *If $U(\sigma, \tau) =$ **Nothing**, then σ and τ are definitely not unifiable, regardless of any type-function reductions*[8].

For example $U(Int, Maybe\ \ a) =$ **Nothing**, because the rigid structure (here Int, $Maybe$) guarantees that they are distinct types, regardless of any substitution for a.

Property 6 (Pre-unifiers)**.** *If $U(\sigma, \tau) =$ **Just** θ and if some ϕ unifies σ and τ (that is, $\phi(\sigma) \sim \phi(\tau)$), then ϕ extends θ.*

A result of **Just** θ indicates that it is *possible* (but not guaranteed) that some substitution ϕ, which extends θ, might make σ and τ equal. For example $U(F\ \ a, Int) =$ **Just** $\varnothing$ because perhaps when $a = Bool$ we might have a family instance $F\ Bool = Int$. Intuitively, θ embodies all the information that U can discover with certainty. We say that θ is a *pre-unifier* of σ and τ and we call U a *pre-unification algorithm*.

These properties are just what we need to refine previous definition of unsound pairwise check:

Definition 7 (Pairwise injectivity with pre-unification)**.** *A pair of equations $F\ \overline{\sigma}_i = \tau_i$ and $F\ \overline{\sigma}_j = \tau_j$, whose variables are disjoint, are pairwise-n-injective iff either*

1. $U(\tau_i, \tau_j) =$ **Nothing**, *or*
2. $U(\tau_i, \tau_j) =$ **Just** θ, *and* $\theta(\sigma_{i\,n}) = \theta(\sigma_{j\,n})$.

As an example, consider $G5$ above. Applying the pairwise injectivity with U test to the two right-hand sides, we find $U([G\ \ a], [Bool])$ = **Just** $\varnothing$, because U immediately returns when it encounters the call $G\ \ a$. That substitution does not make the LHSs identical, so $G5$ is rightly rejected as non-injective.

Now consider this definition:

```
type family G6 a = r | r → a
type instance G6 [a]   = [G a]   -- (1)
type instance G6 Bool = Int      -- (2)
```

Obviously, RHSs of equations (1) and (2) don't unify. Indeed, calling $U([G\ \ a], Int)$ yields **Nothing** and so the pair (1,2) is pairwise-injective. But the injectivity of $G6$ really depends on the injectivity of G: $G6$ is injective iff G is injective. We discover this by performing pairwise test of equation (1) with itself (after freshening). This yields $U(G\ a, G\ a')$. If G is injective U succeeds returning a substitution $[a \mapsto a']$ that makes the LHSs identical, so the pair (1,1) is pairwise-injective. If G is not injective, U still succeeds, but this time with the empty substitution, so the LHSs do not become identical; so (1,1) would not be pairwise-injective, and $G6$ would violate its injectivity condition.

This test is compositional: we can check each definition separately, assuming that the declared injectivity of other definitions holds. In the case of recursive functions, we assume that the declared injectivity holds of calls to the function in its own RHS; and check that, under that assumption, the claimed injectivity holds.

4.2.3 Dealing with Infinity

Our pre-unification algorithm also deals with Awkward Case 3 in Section 4.1, repeated here:

```
type family Z a = r | r → a
type instance Z [a]       = (a, a)
type instance Z (Maybe b) = (b, [b])
```

Classical unification would erroneously declare the RHSs as distinct but, as we saw in Section 4.1, there is a substitution which makes them equal. That is the reason for equation (2) in Figure 2: it conservatively refrains from declaring the types definitely-distinct, and instead succeeds without extending the substitution. Thus $U((a, a), (b, [b]))$ returns the substitution $[a \mapsto b]$ but since that doesn't make the LHSs equal Z is rejected as non-injective.

4.2.4 Closed Type Families

Consider this example of a closed type family:

```
type family Gℤ a = r | r → a where
  Gℤ Int  = Bool
  Gℤ Bool = Int
  Gℤ a    = a
```

The type family $G\mathbb{Z}$ is injective, and we would like to recognize it as such. A straightforward application of the rules we have built up for injectivity will not accept this definition, though. When comparing the first equation against the third, we unify the RHSs, getting the substitution $[a \mapsto Bool]$. We apply this to the LHSs and compare Int with $Bool$; these are not equal, and so the pair of equations appears to be a counter-example to injectivity. Yet, something is amiss: the third equation cannot reduce with $[a \mapsto Bool]$, since the third equation is shadowed by the second one.

This condition is easy to check for. When checking LHSs with a substitution derived from unifying RHSs, we just make sure that if LHSs are different then at least one of the two equations cannot fire after applying the substitution:

Definition 8 (Pairwise injectivity)**.** *A pair of equations $F\ \overline{\sigma}_i = \tau_i$ and $F\ \overline{\sigma}_j = \tau_j$, whose variables are disjoint, are pairwise-n-injective iff either*

1. $U(\tau_i, \tau_j) =$ **Nothing**, *or*
2. $U(\tau_i, \tau_j) =$ **Just** θ, *and*
   (a) $\theta(\sigma_{i\,n}) = \theta(\sigma_{j\,n})$, *or*
   (b) $F\ \theta(\overline{\sigma}_i)$ *cannot reduce via equation i, or*
   (c) $F\ \theta(\overline{\sigma}_j)$ *cannot reduce via equation j*

Note that in an *open* type family, applying a substitution to an equation's LHS will *always* yield a form reducible by that equation,

[8] Readers may be familiar with *apartness* from previous work (Eisenberg et al. 2014). To prove the soundness of our injectivity check, we need $U(\sigma, \tau) =$ **Nothing** to imply that σ and τ are apart.

so the last two clauses are always false. As a result, Definition 8 works for both open and closed type families.

4.3 Soundness

We have just developed a subtle algorithm for checking injectivity annotations. But is the algorithm sound?

Property 9 (Soundness)**.** *If the injectivity check concludes that F is n-injective, then F is n-injective, in the sense of Definition 1.*

In the extended version of this paper (Stolarek et al. 2015), we prove a slightly weaker variant of the property above, and we conjecture (and implement) the full property. The change we found necessary was to omit equation (7) from the statement of the pre-unification algorithm U; this equation allows algorithm U to look under injective type families on the RHS. Without that line, a use of an injective type family in an RHS is treated as is any other type family. Such a modified pre-unification algorithm labels fewer functions as injective. For example, it would reject

$$\textsf{type family } F\ a = r \mid r \to a$$
$$\textsf{type instance } F\ a = Maybe\ (G\ a)$$

even if G were known to be injective.

The full check is quite hard to characterize: what property, precisely, holds of a substitution produced by $U(\tau, \sigma)$? We have said that this substitution is a pre-unifier of τ and σ, but that fact alone is not enough to prove soundness. We leave a full proof as future work.

4.4 Completeness

The injectivity check described here is easily seen to be incomplete. For example, consider the following collection of definitions:

$$\textsf{type family } F\ a = r \mid r \to a$$
$$\textsf{type instance } F\ (Maybe\ a) = G\ a$$
$$\textsf{type instance } F\ [a] = H\ a$$
$$\textsf{type family } G\ a = r \mid r \to a$$
$$\textsf{type instance } G\ a = Maybe\ a$$
$$\textsf{type family } H\ a = r \mid r \to a$$
$$\textsf{type instance } H\ a = [a]$$

The type function F is a glorified identity function, defined only over lists and $Maybe$s. It is injective. Yet, our check will reject it, because it does not reason about the fact that the ranges of G and H are disjoint. Indeed, as argued at the beginning of Section 4, any compositional algorithm will suffer from this problem.

Yet, we would like *some* completeness property. We settle for this one:

Property 10 (Completeness)**.** *Suppose a type family F has equations such that for all right-hand sides τ:*

- *τ is type-family-free,*
- *τ has no repeated use of a variable, and*
- *τ is not a bare variable.*

If F is n-injective, then the injectivity check will conclude that F is n-injective.

Under these conditions, the pairwise injectivity check becomes the much simpler unifying pairwise check of Definition 4, which is enough to guarantee completeness. Note that the conditions mean that Algorithm U operates as a classical unification algorithm (effectively eliminating equations (2), (7), (8), and (9) from the definition of U) and that we no longer have to worry about the single-equation checks motivated by Awkward Case 2 (clause 1 of Definition 2). The proof appears in the extended version of this paper (Stolarek et al. 2015).

4.5 Separate Compilation

The injectivity check must compare *every pair* of equations for a type family F. For closed type families this is straightforward (albeit quadratic) because all the equations are given together. But for open type families the **type instance** declarations may be scattered over many modules. Is the injectivity check consistent with separate compilation?

This issue arises for *all* open type families, regardless of whether they have injectivity annotations: we must always perform a pairwise *compatibility check* (Section 2.1), so it it not a new problem. One way to solve it would be to perform the check only when compiling module *Main*, the module at the top of the module import tree, and then to compare pairwise every family instance in every module transitively imported by *Main*.

That approach would postpone errors too long; for example, a library author might distribute a library with incompatible type-family equations, and only the library clients would get the error message. So in practice GHC makes the compability check when compiling *any* module, checking all equations in that module or the modules it imports. In doing so, GHC can assume that the same checks have already been performed for every imported module, and thereby avoid repeating pairwise comparisons that must have already taken place.

5. Exploiting Injectivity

It is all very well knowing that a function is injective, but how is this fact *useful*? There are two separate ways in which injectivity can be exploited:

Improvement guides the type inference engine, by helping it to fix the values of as-yet-unknown unification variables. Improvement comes in two parts: improvement between "wanted" constraints (Section 5.1) and improvement between wanted constraints and top-level type-family equations (Section 5.2). These improvement rules correspond directly to similar rules for functional dependencies, as we discuss in Section 7.

Decomposition of "given" constraints enriches the set of available proofs, and hence makes more programs typeable (Section 5.3). Unlike improvement, which affects only inference, decomposition requires a small change to GHC's explicitly typed intermediate language, System FC.

5.1 Improvement of Wanted Constraints

Suppose we are given these two definitions:

$$f :: F\ a \to Int$$
$$g :: Int \to F\ b$$

Is the call $(f\ \ (g\ 3))$ well typed? Yes, but it is hard for a type inference engine to determine that this is so without knowing about the injectivity of F. Suppose we instantiate the call to f with a unification variable α, and the call to g with β. Then we have to prove that $F\ \alpha \sim F\ \beta$; we use the term "wanted constraint" for constraints that the inference engine must solve to ensure type safety.

We can certainly solve this constraint if we clairvoyantly unify $\alpha := \beta$. But *the inference engine only performs unifications that it knows must hold*; we say that it performs only *guess-free* unification (Vytiniotis et al. 2011, Section 3.6). Why? Suppose that (in a larger example) we had this group of three wanted constraints:

$$F\ \alpha \sim F\ \beta \quad \alpha \sim Int \quad \beta \sim Bool$$

Then the right thing to do would be unify $\alpha := Int$ and $\beta := Bool$, and hope that $F\ Int$ and $F\ Bool$ reduce to the same thing. Instead unifying $\alpha := \beta$ would wrongly lead to failure.

So, faced with the constraint $F\ \alpha \sim F\ \beta$, the inference engine does *not* in general unify $\alpha := \beta$; so the constraint $F\ \alpha \sim F\ \beta$ is not solved, and hence $f\ (g\ 3)$ will be rejected. But if we knew that F was injective, we can unify $\alpha := \beta$ *without guessing*.

Improvement (a term due to Mark Jones (Jones 1995, 2000)) is a process that adds extra "derived" equality constraints that may make some extra unifications apparent, thus allowing inference to proceed further without having to make guesses. In the case of an injective F, improvement adds $\alpha \sim \beta$, which the constraint solver can solve by unification. In general, improvement of wanted constraint is extremely simple:

Definition 11 (Wanted improvement)**.** *Given the wanted constraint $F\ \overline{\sigma} \sim F\ \overline{\tau}$, add the derived wanted constraint $\sigma_n \sim \tau_n$ for each n-injective argument of F.*

Why is this OK? Because if it is possible to prove the original constraint $F\ \overline{\sigma} \sim F\ \overline{\tau}$, then (by Definition 1) we will also have a proof of $\sigma_n \sim \tau_n$. So adding $\sigma_n \sim \tau_n$ as a new wanted constraint does not constrain the solution space. Why is it beneficial? Because, as we have seen, it may expose additional guess-free unification opportunities that that solver can exploit.

5.2 Improvement via Type Family Equations

Suppose we have the top-level equation

$$\textsf{type instance}\ F\ [a] = a \to a$$

and we are trying to solve a wanted constraint $F\ \alpha \sim (Int \to Int)$, where α is a unification variable. The top-level equation is shorthand for a family of equalities, namely its instances under substitutions for a, including $F\ [Int] \sim (Int \to Int)$. Now we can use the same approach as in the previous section to add a derived equality $\alpha \sim [Int]$. That in turn will let the constraint solver unify $\alpha{:=}[Int]$, and thence solve the wanted constraint. So the idea is to *match* the RHS of the equation against the constraint and, if the match succeeds add a derived equality for each injective argument.

Matters are more interesting when there is a function call on the RHS of the top-level equation. For example, consider $G6$ from Section 4.2.2, when G is injective:

$$\begin{array}{l}\textsf{type family}\ G6\ a = r \mid r \to a\\ \textsf{type instance}\ G6\ [a]\ \ = [G\ a]\\ \textsf{type instance}\ G6\ Bool = Int\end{array}$$

Suppose we have a wanted constraint $G6\ \alpha \sim [Int]$. Does the RHS of the equation, $[G\ a]$, match the RHS of the constraint $[Int]$? Apparently not; but this is certainly the only equation for $G6$ that can apply (because of injectivity). So the argument α must be a list, even if we don't know what its element type is. So we can produce a new derived constraint $\alpha \sim [\beta]$, where β is a fresh unification variable. This expresses *some* information about α (namely that it must be a list type), but not all (the fresh β leaves open what the list element type might be). We might call this *partial improvement*.

Partial improvement is very useful indeed! We can now unify $\alpha := [\beta]$, so the wanted constraint becomes $G6\ [\beta] \sim [Int]$. Now $G6$ can take a step, yielding $[G\ \beta] \sim [Int]$, and decompose to get $G\ \beta \sim Int$. Now the process may repeat, with G instead of $G6$. The crucial points are that (a) the matching step, like the pre-unification algorithm U, behaves specially for type-family calls; and (b) we instantiate any unmatched variables with fresh unification variables. More formally:

Definition 12 (Top-level improvement)**.** *Given:*

- *an equation i of type family F, $F\ \overline{\sigma}_i = \tau_i$, and*
- *a wanted constraint $F\ \overline{\sigma}_0 \sim \tau_0$*

such that

- $M(\tau_i, \tau_0) =$ **Just** θ*, and*
- $F\ \theta(\overline{\sigma}_i)$ *can reduce via equation i*

then define θ' by extending θ with $a \mapsto \alpha$, for every a in $\overline{\sigma}_i$ that is not in $dom(\theta)$, where α is a fresh unification variable; and add a derived constraint $\theta'(\sigma_{i\,n}) \sim \sigma_{0\,n}$, for every n-injective argument of F.

Here M is defined just like U in Figure 2, except lacking equations (4) and (9). That is, M does one-way matching rather than two-way unification. (We assume that the variables of the two arguments to M do not overlap.)

5.3 Decomposing Given Equalities

Consider the following function, where F is an injective type family:

$$\begin{array}{l}fid :: (F\ a \sim F\ b) \Rightarrow a \to b\\ fid\ x = x\end{array}$$

Should that type-check? Absolutely. We assume that $F\ a \sim F\ b$, and by injectivity (Definition 1), we know that $a \sim b$. But, arranging for GHC to compile this requires a change to System FC.

In FC, all type abstractions, applications, and casts are explicit. FC code uses a proof term, or *coercion*, that witnesses the truth of each equality constraint. In FC, *fid* takes an argument coercion $c :: F\ a \sim F\ b$, but needs a coercion of type $a \sim b$ to cast $x :: a$ to the desired result type b. With our proposed extension the FC code for *fid* will look like this:

$$\begin{array}{l}fid :: \forall\ a\ b.\ (F\ a \sim F\ b) \Rightarrow a \to b\\ fid = \Lambda a\ b \to \lambda(c :: F\ a \sim F\ b)\ (x :: a) \to x \triangleright (\mathbf{nth}^0\ c)\end{array}$$

The coercion $(\mathbf{nth}^0\ c)$ is a proof term witnessing $a \sim b$. Using $\mathbf{nth}$ to decompose a type family application is the extension required to FC, as we discuss next.

5.3.1 Adding Type Family Injectivity to FC

To a first approximation, System FC is Girard's System F, enhanced with equality coercions. That is, there is a form of expression $e \triangleright \gamma$ that casts e to have a new type, as shown by the following typing rule:

$$\frac{\Gamma \vdash e : \tau_1 \qquad \Gamma \vdash \gamma : \tau_1 \sim \tau_2}{\Gamma \vdash e \triangleright \gamma : \tau_2}\ \textsc{Tm\_Cast}$$

The unusual typing judgement $\Gamma \vdash \gamma : \tau_1 \sim \tau_2$ says that γ is a proof, or witness, that type τ_1 equals type τ_2.

Coercions γ have a variety of forms, witnessing the properties of equality required from System FC. For example, there are forms witnessing reflexivity, symmetry, and transitivity, as well as congruence of application; the latter allows us to prove that types $\tau_1\ \tau_2$ and $\sigma_1\ \sigma_2$ are equal from proofs that $\tau_1 \sim \sigma_1$ and $\tau_2 \sim \sigma_2$.

The coercion form that concerns us here is the one that witnesses injectivity. In FC the rule looks thus:

$$\frac{\Gamma \vdash \gamma : H\ \overline{\tau} \sim H\ \overline{\sigma}}{\Gamma \vdash \mathbf{nth}^i\ \gamma : \tau_i \sim \sigma_i}\ \textsc{Co\_Nth}$$

In this rule, H is a type constant (such as $Maybe$ or $(\to)$), all of which are considered to be injective in Haskell. The coercion $\mathbf{nth}^i\ \gamma$ witnesses this injectivity by proving equality among arguments from the equality of the applied datatype constructor.

To witness injective type families, we must add a new rule as follows:

$$\frac{\begin{array}{l}\Gamma \vdash \gamma : F\ \overline{\tau} \sim F\ \overline{\sigma}\\ F \text{ is } i\text{-injective}\end{array}}{\Gamma \vdash \mathbf{nth}^i\ \gamma : \tau_i \sim \sigma_i}\ \textsc{Co\_NthTyFam}$$

In this rule, F is a type family. We can now extract an equality among arguments from the equality proof of the type family applications.

5.3.2 Soundness of Type Family Injectivity

Having changed GHC's core language, we now have the burden of proving our change to be type safe. The key lemma we must consider is the consistency lemma. Briefly, the consistency lemma states that, in a context with no equality assumptions, it is impossible to prove propositions like $Int \sim Bool$, or $(a \to b) \sim IO\ ()$. With the consistency lemma in hand, the rest of the proof of type safety would proceed as it has in previous publications, for example Breitner et al. (2014).

Even stating the key lemmas formally would require diving deeper into System FC than is necessary here; the lemmas and their proofs appear in the extended version of this paper (Stolarek et al. 2015).

5.4 Partial Type Functions

Both open and closed type families may be *partial*; that is, defined on only part of their domain. For example, consider this definition for an injective function F:

$$\begin{array}{l}\textsf{type family}\ F\ a = r \mid r \to a\\ \textsf{type instance}\ F\ Int = Bool\\ \textsf{type instance}\ F\ [a] = a \to a\end{array}$$

The type $F\ [Char]$ is equal to $Char \to Char$, by the second instance above; but $F\ Bool$ is equal only to itself since it matches no equation. Nevertheless, F passes our injectivity test (Section 4).

You might worry that partiality complicates our story for injectivity. If we had a wanted constraint $F\ Bool \sim F\ Char$, our improvement rules would add the derived equality $Bool \sim Char$, which is manifestly insoluble. But nothing has gone wrong: the original wanted constraint was *also* insoluble (that is, we could not cough up a coercion that witnesses it), so all the derived constraint has done is to make that insolubility more stark.

In short, the fact that type functions can be partial does not gum up the works for type inference.

6. Injectivity in the Presence of Kind Polymorphism

Within GHC, kind variables are treated like type variables: type family arguments can include both kinds and types. Thus type families can be injective not only in type arguments but also in kind arguments. To achieve this we allow kind variables to be mentioned in the injectivity condition, just like type variables. Moreover, if a user lists a type variable b as injective, then all kind variables mentioned in b's kind are also marked as injective. For example:

$$\begin{array}{l}\textsf{type family}\ G\ (a :: k1)\ (b :: k2)\ (c :: k1)\\ \quad = (r :: k3) \mid r \to b\ k1\\ \textsf{type instance}\ G\ Maybe\ Int\ \ (Either\ Bool) = Char\\ \textsf{type instance}\ G\ IO\ \ \ Int\ \ [\,]\ \ \ \ \ \ \ \ = Char\\ \textsf{type instance}\ G\ Either\ Bool\ (\to)\ \ \ \ \ \ = Maybe\end{array}$$

The injectivity annotation on G states that it is injective in b – and therefore also in b's kind $k2$ – as well as kind $k1$, which is the kind of both a and c. We could even declare $k3$ as injective – the return kind is also an input argument to a type family.

To support injectivity in kinds our pre-unification algorithm U needs a small adjustment to make it kind-aware – see modified equations (2) and (3) in Figure 3. Other definitions described in Sections 4 and 5 remain unchanged.

In Haskell source, in contrast to within GHC, kind arguments are treated quite separately from type arguments. Types are always explicit, while kinds are always implicit. This can lead to some surprising behaviour:

$$\begin{array}{l}\textsf{type family}\ P\ (a :: k0) = (r :: k1) \mid r \to a\\ \textsf{type instance}\ P\ '[\,] = '[\,]\end{array}$$

At first glance, P might look injective, yet it is not. Injectivity in a means injectivity also in $k0$. But the argument a and result r can have different kinds and so $k0$ is not determined by r. This becomes obvious if we write kind arguments explicitly using a hypothetical syntax, where the kind arguments are written in braces:

$$\textsf{type instance}\ P\ \{k0\}\ \{k1\}\ ('[\,]\ \{k0\}) = ('[\,]\ \{k1\})$$

The syntax $('[\,]\ \{k\})$ indicates an empty type-level list, holding elements of kind k[9]. It is now clear that $k0$ is not mentioned anywhere in the RHS, and thus we cannot accept it as injective.

$$\begin{array}{l}U(a : \kappa_1, \tau : \kappa_2)\ \theta = U(\kappa_1, \kappa_2)\ \theta \qquad\qquad a \in ftv(\theta(\tau))\\ U(a : \kappa_1, \tau : \kappa_2)\ \theta = U(\kappa_1, \kappa_2)\ ([a \mapsto \theta(\tau)] \circ \theta)\ a \notin ftv(\theta(\tau))\end{array}$$

Figure 3. Modified equations (2) and (3) from Figure 2 that make the pre-unification algorithm U kind-aware.

7. Functional Dependencies

Injective type families are very closely related to type classes with *functional dependencies* (Jones 2000), which have been part of GHC for many years. Like injectivity, functional dependencies appear quite simple, but are remarkably subtle in practice (Sulzmann et al. 2007).

Functional dependencies express a type level function as a *relation*. For example, here are type-level functions F and G expressed using functional dependencies (on the left) and type families (on the right):

$$\begin{array}{ll}\textsf{class}\ F\ a\ r \mid a \to r & \textsf{type family}\ F\ a = r\\ \textsf{instance}\ F\ [a]\ (Maybe\ a) & \textsf{type instance}\ F\ [a] = Maybe\ a\\ \textsf{instance}\ F\ Int\ Bool & \textsf{type instance}\ F\ Int = Bool\\ \textsf{class}\ G\ a\ r \mid a \to r & \textsf{type family}\ G\ a = r\\ f :: F\ a\ r \Rightarrow a \to r & f :: a \to F\ a\end{array}$$

To express that F and G are injective using functional dependencies, one adds an additional dependency:

$$\begin{array}{l}\textsf{class}\ F\ a\ r \mid a \to r, r \to a\\ \textsf{class}\ G\ a\ r \mid a \to r, r \to a\end{array}$$

This syntax motivates our choice of syntax for injectivity annotations (Section 3.2), and our injectivity check mirrors precisely the consistency checks necessary for functional dependencies (Jones 2000).

In Section 4.2.2 we discussed the issues that arise when a call to an injective type family G appears in the RHS of a **type instance**, such as:

$$\textsf{type instance}\ F\ (Maybe\ a) = [G\ a]$$

Precisely the same set of issues arises with functional dependencies, where the instance declaration would look like:

$$\textsf{instance}\ G\ a\ rg \Rightarrow F\ (Maybe\ a)\ [rg]$$

This instance declaration would fail the *coverage condition* of (Jones 2000); in effect, Jones does not allow function calls on the RHS. This restriction was lifted by Sulzmann *et al*, via the *liberal coverage condition* (Sulzmann et al. 2007), in essentially the same way that we do.

[9] You can see similar output from GHC – without the braces – if you use `-fprint-explicit-kinds`.

Using "improvement" to guide type inference (Section 5), including the partial improvement of Section 5.2, was suggested by Mark Jones for his system of qualified types (Jones 1995), and was absolutely essential for effective type inference with functional dependencies (Jones 2000). Indeed, the improvement rules of Section 5 correspond precisely to the improvement rules for functional dependencies (Sulzmann et al. 2007).

7.1 Advantages of Type Families

A superficial but important advantage of type families is simply that they use functional, rather than relational, notation, thus allowing programmers to use same programming style at the type level that they use at the term level. Recognizing this, Jones also proposes some syntactic sugar to make the surface syntax of functional dependencies more function-like (Jones 2008). Syntactic sugar always carries a price, of course: since the actual types will have quantified constraints that are not visible to the programmer, the compiler has to work hard to express error messages, inferred types, and so on, in the form that the programmer expects.

A more substantial difference is that type families are fully integrated into System FC, GHC's typed intermediate language. Consider, for example:

$$\begin{array}{l}\textsf{type instance}\ F\ Int = Bool\\ \textsf{data}\ T\ a\ \textsf{where}\ \{\,MkT :: F\ a \to T\ a\,\}\\ f :: T\ Int \to Bool\\ f\ (MkT\ x) = \textsf{not}\ x\end{array}$$

This typechecks fine. But with functional dependencies we would write

$$\begin{array}{l}\textsf{class}\ F\ a\ r \mid a \to r\\ \textsf{instance}\ F\ Int\ Bool\\ \textsf{data}\ T\ a\ \textsf{where}\ \{\,MkT :: F\ a\ r \Rightarrow r \to T\ a\,\}\end{array}$$

and now the definition of f would be rejected because r is an existentially captured type variable of MkT. One could speculate on a variant of System FC that accommodated functional dependencies, but no such calculus currently exists.

7.2 Advantages of Functional Dependencies

Functional dependencies make it easy to specify more complex dependencies than mere injectivity. For example[10]:

$$\begin{array}{l}\textsf{data}\ Nat = Zero \mid Succ\ a\\ \textsf{class}\ Add\ a\ b\ r \mid a\ b \to r, r\ a \to b\\ \textsf{instance}\qquad\qquad Add\ Zero\quad b\ b\\ \textsf{instance}\ (Add\ a\ b\ r) \Rightarrow Add\ (Succ\ a)\ b\ (Succ\ r)\end{array}$$

Note the dependency "$r\ a \to b$", which says that the result and first argument (but not the result alone) are enough to fix the second argument. This dependency leads to an improvement rule: from the wanted constraint $(Add\ s\ t1) \sim (Add\ s\ t2)$, add the derived equality $t1 \sim t2$.

Our design can similarly be extended, by writing:

$$\begin{array}{l}\textsf{type family}\ AddTF\ a\ b = r \mid r\ a \to b\ \textsf{where}\\ \quad AddTF\ Zero\quad b = b\\ \quad AddTF\ (Succ\ a)\ b = Succ\ (AddTF\ a\ b)\end{array}$$

The check that the injectivity annotation is sound would be an extension of Definitions 2 and 8, and the improvement rule would mimic the one for functional dependencies. However, this remains as future work: we have not yet extended the metatheory or implementation to accommodate it.

[10] Here Nat is being used as a kind, using the $DataKinds$ extension (Yorgey et al. 2012).

7.3 Summary

So, are type families merely functional dependencies in a different guise? At a fundamental level, yes: they both address a similar question in a similar way. But it is always illuminating to revisit an old landscape from a new direction, and we believe that is very much the case here, especially since the landscape of functional dependencies is itself extremely subtle (Sulzmann et al. 2007). Understanding the connection better is our main item of further work. For example:

- Adding richer functional dependencies to type families (Section 7.2) is an early priority.
- Could we take advantage of the metatheory of functional dependencies to illuminate that of type families; or vice versa?
- What would a version of System FC that truly accommodated functional dependencies look like?
- Could closed type families move beyond injectivity and functional dependencies by applying closed-world reasoning that derives solutions of arbitrary equalities, provided a unique solution exists? Consider this example:

  $$\begin{array}{l}\textsf{type family}\ J\ a\ \textsf{where}\\ \quad J\ Int\ \ = Char\\ \quad J\ Bool\ \ = Char\\ \quad J\ Double = Float\end{array}$$

  One might reasonably expect that if we wish to prove $(J\ a \sim Float)$, we will simplify to $(a \sim Double)$. Yet GHC does not do this as neither injectivity nor functional dependencies can discover this solution.

8. Other Related Work

8.1 Injectivity for the Utrecht Haskell Compiler

Implementing injective type families for the Utrecht Haskell Compiler was proposed by Serrano Mena (2014). These ideas were not developed further or implemented[11]. Thus, to our best knowledge, our work is the first theoretical and practical treatment of injectivity for Haskell.

8.2 Injectivity in Other Languages

The Agda (Norell 2007) compiler is able to infer *head injectivity*[12], a notion weaker than the injectivity presented in this paper. For a function f, if the right-hand sides of all clauses of f immediately disunify, then f is called head-injective or *constructor-headed*. "Immediately disunify" means that the outer-most constructors in the RHSs are distinct. Knowledge that a function is head-injective can then be used to generate improvements in the same way it is used in our solution. Our solution is more powerful: it recurs over identical constructors, allows type families in RHSs, and permits declaring injectivity only in some arguments.

Other dependently-typed languages like Coq (The Coq development team 2014) or Idris (Brady 2013) do not provide any special way of declaring that a function is injective. In these languages the user can prove injectivity of a function using mechanisms provided by the language (e.g. tactics) and appeal to injectivity explicitly whenever this property is required to make progress during type checking. We believe that these languages could benefit from approach developed here – our results should carry over to these other languages nicely.

[11] Based on private correspondence with Alejandro Serrano Mena.

[12] Based on private correspondence with Andreas Abel.

8.3 Injectivity of Term-Rewriting Systems

Haskell type families share much with traditional term-rewriting systems (TRSs). (For some general background on TRSs, see Baader and Nipkow (1998).) In particular, Haskell type family reduction forms a deterministic constructor term-rewriting system. There has been some work done on checking TRSs for injectivity, for example that of Nishida and Sakai (2010). Their work appears to be the state-of-the-art in the term-rewriting community. Although a close technical comparison of our work to theirs is beyond the scope of this paper, Nishida and Sakai restrict their injectivity analysis to total, terminating systems. Our work also considers partial and non-terminating functions.

9. Conclusion

With this work, we give users a new tool for more expressive type-level programming, and one that solves practical problems arising in the wild (Section 2). It fills out a missing corner of GHC's support for type-level programming, and gives an interesting new perspective on functional dependencies (Section 7).

Our compositional approach for determining injectivity of functions defined by pattern matching may be of more general utility.

Acknowledgements

We gratefully acknowledge helpful feedback from Iavor Diatchki, Martin Sulzmann, Dimitrios Vytiniotis, and Stephanie Weirich. The third author gratefully acknowledges support from a Microsoft Research Graduate Student Fellowship. This material is based upon work supported by the National Science Foundation under Grant No. 1116620.

References

F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, New York, NY, USA, 1998. ISBN 0-521-45520-0.

E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5): 552–593, Sept. 2013.

J. Breitner, R. A. Eisenberg, S. Peyton Jones, and S. Weirich. Safe zero-cost coercions for Haskell. In *ACM SIGPLAN International Conference on Functional Programming*, pages 189–202, Sept. 2014.

M. M. T. Chakravarty, G. Keller, and S. Peyton Jones. Associated type synonyms. In *ACM SIGPLAN International Conference on Functional Programming*, 2005a.

M. M. T. Chakravarty, G. Keller, S. Peyton Jones, and S. Marlow. Associated types with class. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2005b.

J. Cheney and R. Hinze. First-class phantom types. Technical report, Cornell University, 2003.

R. A. Eisenberg, D. Vytiniotis, S. Peyton Jones, and S. Weirich. Closed type families with overlapping equations. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2014.

M. P. Jones. Simplifying and improving qualified types. In *ACM Conference on Functional Programming and Computer Architecture*, 1995.

M. P. Jones. Type classes with functional dependencies. In *European Symposium on Programming*, 2000.

M. P. Jones. Language and program design for functional dependencies. In *ACM Haskell Symposium*. ACM, 2008.

S. Lindley and C. McBride. Hasochism: the pleasure and pain of dependently typed Haskell programming. In *ACM SIGPLAN Haskell Symposium*, pages 81–92, 2013. .

N. Nishida and M. Sakai. Proving injectivity of functions via program inversion in term rewriting. In M. Blume, N. Kobayashi, and G. Vidal, editors, *Functional and Logic Programming*, volume 6009 of *Lecture Notes in Computer Science*, pages 288–303. Springer, 2010.

U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *ACM SIGPLAN International Conference on Functional Programming*, 2006.

A. Serrano Mena. Branching, disjointness and injectivity for OutsideIn(X). ICFP 2014 poster, Sept. 2014.

J. Stolarek, S. Peyton Jones, and R. A. Eisenberg. Injective type families for Haskell (extended version). Technical report, Politechnika Łódzka, 2015. URL `http://ics.p.lodz.pl/~stolarek/_media/pl:research:stolarek_peyton-jones_eisenberg_injectivity_extended.pdf`.

M. Sulzmann, G. Duck, S. Peyton Jones, and P. Stuckey. Understanding functional dependencies via constraint handling rules. *Journal of Functional Programming*, 17:83–129, 2007.

The Coq development team. The Coq Proof Assistant reference manual, version 8.4. Technical report, TypiCal Project (formerly LogiCal), Nov. 2014.

D. Vytiniotis, S. Peyton Jones, T. Schrijvers, and M. Sulzmann. OutsideIn(X): Modular type inference with local assumptions. *Journal of Functional Programming*, 21(4-5):333–412, Sept. 2011.

B. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving Haskell a promotion. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation*, 2012.

A. Popularity of Selected Language Extensions for Type-Level Programming

In Section 1 we made a claim that type families are the most popular language extension for type-level programming in Haskell. That claim is based on analysis of Hackage, Haskell's community package database. We were interested in usage of five language extensions that in our opinion add the most powerful features to type-level language: *TypeFamilies*, *GADTs*, *FunctionalDependencies*, *DataKinds* and *PolyKinds*. To measure their popularity we downloaded all packages on Hackage (per list available at `https://hackage.haskell.org/packages/names`). Then we used the *grep* program to search each package directory for appearances of strings naming the given language extensions. This located language extensions enabled both in *.cabal* files and with *LANGUAGE* pragmas. The exact obtained numbers are reported in Table 1.

Table 1. Popularity of selected type-level programming language extensions.

Language extension	no. of using packages
TypeFamilies	1092
GADTs	612
FunctionalDependencies	563
DataKinds	247
PolyKinds	109

Downside of this approach is that it can give false positives by finding strings without considering their context inside the source code. A good example of when this happens is haskell-src-exts package that does not use any of the above extensions but mentions them in the parser source code.

All measurements were conducted on a copy of Hackage obtained on 19th February 2015.