

Towards Open Type Functions for Haskell

Tom Schrijvers^{*1}, Martin Sulzmann², Simon Peyton-Jones³, and Manuel Chakravarty⁴

¹ K.U.Leuven, Belgium (tom.schrijvers@cs.kuleuven.be)

² National University of Singapore (sulzmann@comp.nus.edu.sg)

³ Microsoft Research Cambridge, UK (simonpj@microsoft.com)

⁴ University of New South Wales (chak@cse.unsw.edu.au)

Abstract. We report on an extension of Haskell with type(-level) functions and equality constraints. We illustrate their usefulness in the context of phantom types, GADTs and type classes. Problems in the context of type checking are identified and we sketch our solution: a decidable type checking algorithm for a restricted class of type functions. Moreover, functional dependencies are now obsolete: we show how they can be encoded as type functions.

1 Introduction

Experimental languages such as ATS [6], Cayenne [1], Chameleon [25], Epigram [15] and Omega [21] equip the programmer with various forms of “type functions” to write entire programs on the level of types. In the context of Haskell, there are two distinct languages extensions that support such type-level computation: *functional dependencies* which are well established [12], and *associated types* which are a more recent experiment [5]. In this paper, we make the following contributions:

- We generalise the so-called “associated type synonyms” [5] by decoupling them from `class` declarations, thereby allowing us to define stand-alone type functions (Section 2). We give examples which show the usefulness of stand-alone type functions in combination with GADTs and phantom types.
- It turns out that pure type *inference* for our extended language is very easy. However, in the presence of user-supplied type signatures (which are ubiquitous in Haskell) and GADTs, the type *checking* problem becomes unexpectedly hard. We identify the problem and sketch our solution (Section 3). This is the main technical contribution of the paper.
- We show that type functions are enough to express all programs involving functional dependencies, although the reverse is problematic (Section 4). Other related work is discussed in Section 5.

For space reasons, and because it reports work in progress, this paper is entirely informal. We have much formal material in an accompanying draft technical report [20].

* Post-doctoral researcher of the Fund for Scientific Research - Flanders.

2 Informal overview

We begin informally, by giving several examples that motivate type functions, and show what can be done with them. Notably, we have found three uses of type functions: in combination with type classes, in combination with GADTs and even in the basic Hindley/Milner type system.

2.1 Type classes and type functions

The original paper on functional dependencies [12] presented the following class of collections:

```
class Collects c e | c -> e where
  empty  :: c
  insert :: e -> c -> c
  toList :: c -> [e]
instance Collects BitSet Char where ...
instance Eq e => Collects c [c] where ...
```

The notation “`| c -> e`” means “the collection type `c` determines the element type `e`”. The two instance declarations explain that the collection of type `BitSet` has elements of `Char` elements; and a collection of type `[e]` has elements of type `e`. The “...” parts give the implementations of the methods `empty`, `insert`, etc.

Using our proposed type-function extension we would re-express the example as follows:

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

type instance Elem BitSet = Char
instance Collects BitSet where ...

type instance Elem [e] = e
instance Eq e => Collects c [c] where ...
```

The type class now has only one parameter, `c`. A new *type family* `Elem` is defined, using a `type family` declaration. We think of `Elem` as a function from the collection type to the element type, and indeed often refer to it as a “type function”, although the term “function” is so heavily used that we use “type family” when we want to be precise. The types of the class methods should now be self-explanatory; indeed, they are more perspicuous than before.

Each instance declaration now has two related parts. First, we add an equation to the definition of `Elem`, using a `type instance` declaration. Second, we have a perfectly ordinary Haskell `instance` declaration.

One might wonder whether functional dependencies are more expressive than type functions, or vice versa, and we discuss that in Section 4.

2.2 Plain type functions

The main benefit of decoupling the type functions from type classes is that the former can now be used independently. (We still offer the syntax for associated type synonyms proposed in [5], but it is purely syntactic sugar.) For example, here is how we might write a library that manipulates lengths, areas, volumes, and so on:

```
data Z      -- Peano numbers
data S a    -- at the level of types

newtype Val u = V Float

type Scalar = Val Z
type Length = Val (S Z)
type Area   = Val (S (S Z))
type Volume = Val (S (S (S Z)))

addVal :: Val u -> Val u -> Val u
addVal (V v1) (V v2) = V (v1+v2)

mulVal :: Val u1 -> Val u2 -> Val (Sum u1 u2)
mulVal (V v1) (V v2) = V (v1*v2)
```

The phantom-type parameter `u` keeps track (statically) of the units of the value [11]. The idea is that `u` will be instantiated by a Peano-number representation of the dimension of the number, as suggested by the (ordinary, Haskell) type synonym declarations of `Scalar` etc. The signature of `addVal` specifies that it can only add two values of the same units.

The signature of `mulVal` is more interesting, because the dimension of the result is the sum of the dimensions of its argument — for example, multiplying a `Length` by an `Area` gives a `Volume`. So we need a type-level computation, expressed using the type function `Sum`:

```
type family Sum n m
type instance Sum Zero x      = x
type instance Sum (Succ x) y = Succ (Sum x y)
```

Notice that no type classes are involved here. The same program can be written using functional dependencies, but only by bringing in a type class (with no methods), and only by using a more relational notation:

```
mulVal :: Sum u1 u2 r => Val u1 -> Val u2 -> Val r
```

The example can readily be extended to handle multiple units (e.g. time as well as length).

This encoding does not express the fact that `Val` should *only* be applied to compositions of `S` and `Z`. It would be better to express this idea in the kinds thus:

```
datakind Nat = Z | S Nat
```

```
newtype Val (u::Nat) = V Float
```

Not only is this more explicit, but it also allows us to check that we have provided all the equations for `Sum`, and permits induction over `Nat`. In effect, `Sum` is a *closed* type function whereas `Elem` was an *open* one. In this paper we concentrate on open functions, and leave the exploitation of closed-ness for future work.

2.3 GADTs and type functions

Generalised Algebraic Data Types (GADTs) are extremely useful for expressing rich data structure invariants at the type level. A well-known example is that of length-indexed lists, or *vectors* for short:

```
data Vector el len where
  Nil  :: Vector el Z
  Cons :: el  -> Vector el len -> Vector el (S len)
```

where we use the type encoding of the natural numbers from the previous section.

With vectors we can easily avoid some of the pitfalls of ordinary lists. Consider the well-known Haskell function `zip :: [a] -> [b] -> [(a,b)]` for pairing up the corresponding elements in two lists. It has an annoying corner case: when the lengths of the two lists are not matched, then the trailing elements of the longer list are simply and silently discarded. With vectors we can easily rule out this corner case at compile time. Consider the definition of `vzip`, a `zip` for vectors:

```
vzip :: Vector a len -> Vector b len -> Vector (a,b) len
vzip Nil Nil = Nil
vzip (Cons x xs) (Cons y ys) = Cons (x,y) (vzip xs ys)
```

Observe that the length type parameter of both input lists is identical. This means that the type checker verifies for every call `vzip as bs` whether the vectors `as` and `bs` have the same length. If not, the program is rejected.

For length-indexing to be useful, we should be able to express the impact of list transformations on the length. Unfortunately, without resorting to overly complicated⁵ type classes with functional dependencies, Haskell's type system does not allow us to express even the most basic of transformations.

Concatenation of vectors is a good example. While this implementation is easy enough to write:

```
vconcat Nil          l  = l
vconcat (Cons x xs) ys = Cons x (vconcat xs ys)
```

and, like `mulVal`, its signature involves a type-level computation;

```
vconcat :: Vector e n -> e Vector e m -> Vector e (Sum n m)
```

This function `Sum` is a type-level function, defined in the previous section.

⁵ and rather ill-understood

2.4 Equality constraints

Suppose that we want to write a function `merge` that adds all the elements of one collection to another collection. It cannot have type

```
merge :: (Collects c1,Collects c2) => c1 -> c2 -> c2
```

because not all collections have the same element types. On the the other hand, it is over-restrictive to write

```
merge :: (Collects c) => c -> c -> c
```

because it is perfectly OK to merge collections of different types *provided their element types are the same*. For example one could merge a `BitSet` with a `[Char]` because they both have element type `Char`.

The way to achieve this is to use an *equality constraint*:

```
merge :: (Collects c1,Collects c2,Elem c1 ~ Elem c2) =>
         c1 -> c2 -> c2
```

where constraint “`Elem c1 ~ Elem c2`” says that `c1` and `c2` must only be instantiated to types for which `Elem c1` and `Elem c2` are equal. These equality constraints are, in fact, quite familiar from GADTs. Recall the definition of `Vector` from the previous section:

```
data Vector el len where
  Nil  :: Vector el Z
  Cons :: el -> Vector el len -> Vector el (S len)
```

One way to think of `Cons` is that it has type

```
Cons :: (slen ~ S len) => el -> Vector el len -> Vector el slen
```

Our new design allows arbitrary type equalities to be specified in a type signature, with GADTs as a useful special case. In [5] a number of restrictions are imposed on the form of equality constraints. We do not impose any restrictions; even constraints that do not involve any type functions are allowed, e.g. `Int ~ Bool`.

2.5 Summary

In general, a type function is introduced by a top-level `type family` declaration. An optional kind signature may be used for both the argument types and the result type; for example:

```
type family MonadRef (monad :: * -> *) :: (* -> *)
```

Otherwise, these kinds are assumed to be `*`.

Like a regular Haskell 98 type synonym, a type function has an *arity*, given by the number of named arguments to the left of the “`::`”. For example `MonadRef`

has arity 1, even though it has kind “*->*->*”. Like regular type synonyms, type-function applications must be *saturated*: they must be supplied with at least as many type as prescribed by their arity. Again like type synonyms, over-application is of course allowed, e.g. `MonadRef IO Int`.

Unlike regular type synonyms, however, type functions are open functions, whose definition is extended by `type instance` declarations; for example:

```
type instance F [a] k = (a,k)
```

The part to the left of the “=” is called the *definition head* and the part to the right the *definition body*. The head must have exactly as many type parameters as the arity of the type function.

In order to ensure modularity, consistency of the type function definition and termination of type inference, a number of conditions must be imposed on the instances:

1. Instance heads must not overlap.
2. Type function applications in the body must be smaller than the head.
3. Type function applications in the body must not occur inside other type function applications.

We return in more detail on these conditions when we discuss type checking.

Just as Haskell has data types as well as type synonyms, we also support *data type families* as well as type functions. For example:

```
data family GMap k v
data instance GMap Int v = GI (Map.Map Int v)
data instance GMap (a, b) v = GP (GMap a (GMap b v))
```

Like a `type instance`, there may be many `data instance` declarations for each data family, each having a different type pattern to the left of the “=”. The rest of the declaration is just like a regular Haskell data type declaration: it defines one or more constructors. (They can even be GADTs!)

Data type families are really only useful in association with type classes; we refer the reader to [4] for details. In contrast to type functions, it is extremely straightforward to add data type families to the type inference engine, and we do not discuss them further here.

3 Technical challenges

To add type functions to Haskell we must explain how to adapt the type inference engine to accommodate them. Parts of this turned out to be very easy but, somewhat to our surprise, other parts were much harder than we anticipated. One of the main contributions of this paper is to identify just what is hard, although we have space only to sketch our solution.

3.1 The easy part: type inference

Consider the problem of doing pure type inference (i.e. with no types declared by the programmer) in the presence of type functions.

This is an easy problem. Recall that the `type instance` declarations are restricted (Section 2.5) so that they can be regarded as a left-to-right rewrite system that is (a) confluent and (b) terminating. Type inference is conventionally done using unification (see [18] for a tutorial). When type functions are added, we modify the unifier so that when it tries to unify two types, it first *normalises* them using the rewrite rules.

If performed too early, this normalisation may get “stuck”. For example, consider inferring the type for

```
\c -> (insert 'x' c, length c)
```

where `insert` was defined in Section 2.1, and `length` has its usual type:

```
insert :: Collects c => Elem c -> c -> c
length :: [a] -> Int
```

Initially, type inference assigns an unknown type α to `c`. The `insert` call requires us to unify `Char` (the type of `'x'`) with `Elem α` (the argument type of `insert`). Since we do not know what α is, normalisation gets stuck. But all is well, because “later”, the call `length c` forces α to be unified with `[β]`; and now the stuck normalisation can proceed, rewriting `Elem [β]` to β .

So all we need is a way to suspend stuck unifications, and try them again later. That is, we must gather as-yet-unsatisfied equality constraints from the term, and attempt to solve them later. Happily, Haskell already requires us to gather type-class constraints from the term, so all the plumbing is already in place.

All that remains is to consider generalisation. Consider the definition

```
f = \c -> insert 'x' c
```

When we come to generalise `f`, the stuck unification is still stuck! But that is easy: just as we abstract over *type class* constraints in this situation, so we abstract over *equality* constraints, to give the type

$$f :: \forall a. (\text{Collects } a, \text{Elem } a \sim \text{Char}) \implies a \rightarrow a$$

3.2 The hard part: type checking

Alas, we cannot live with type inference alone. Type checking is necessary as well, for a number of reasons:

- Programmers want to write signatures, as a form of specification or documentation of their program.
- Full type inference is infeasible for a number of type system features, notably for GADTs [24].

Consider again the `vconcat` function:

```

vconcat :: Vector e n -> Vector e m -> Vector e (Sum n m)
vconcat Nil          1    = 1
vconcat (Cons x xs) ys = Cons x (vconcat xs ys)

```

Let us focus on the first equation alone, the case for `Nil`. We know that `1` is of type `Vector e m`. The program type checks if we can show that `1` is also of type `Vector e (Sum n m)`. If we drop the identical parts, this boils down to showing that `m` equals `Sum n m`. How can we establish this equality? The pattern match `Nil` makes available the (local) assumption $n \sim Z$. So we want to deduce that

$$n \sim Z \implies m \sim \text{Sum } n \ m$$

And this holds, of course, because we can make use of the top-level type-function equations for `Sum`:

$$(\forall ys. \text{Sum } Z \ ys \sim ys), \\ (\forall xs, ys. \text{Sum } (S \ xs) \ ys \sim S \ (\text{Sum } \ xs \ ys)) \models n \sim Z \implies m \sim \text{Sum } n \ m$$

Similar reasoning applies to the `Cons` case.

In general, type checking is reduced to an entailment check among type equations with respect to an equational theory:

$$E_t \models E_g \implies E_w$$

where

- E_t (top-level equations) refers to the type function theory, i.e. the top-level type function definitions. These equations may involve universal quantification; e.g. $\forall ys. \text{Sum } Z \ ys \sim ys$.
- E_g are the given equations arising from type annotations and GADT pattern matchings, for example $n \sim Z$. These equations are over monotypes, with no universal quantification.
- E_w are the wanted equations arising out of expressions, for example $m \sim \text{Sum } n \ m$. Again, the equations are over monotypes.

3.3 The type checking strategy

The type inference strategy was to use the top-level equations E_t to normalise the wanted constraints E_w . But we cannot do this for type checking, *because the additional given constraints E_g do not necessarily form a terminating, confluent rewrite system*, particularly when combined with E_t :

1. They are not properly oriented to ensure termination. E.g. the TRS formed by top-level equation `F Bool = Int` and given equation `Int ~ F Bool` is clearly looping.
2. They may well be inconsistent (i.e. non-confluent) with respect to each other or the top-level equations. E.g. the TRS formed by top-level equation `F Bool = Int` and given equation `F Bool ~ Char` is not consistent.

The solution of these issues is to transform the given equations E_g into an equivalent set of equations E'_g that *does* satisfy all the necessary properties. In TRS-terminology, the problem of finding an such an E'_g is known as the *completion* problem.

Unfortunately, there is no off-the-shelf completion algorithm that suits our needs. Existing completion procedures are either undecidable [3] or restricted to systems of ground equations [19]. What we require is a completion algorithm that is (1) decidable and (2) takes into account the non-ground top-level equations. In addition, we want to exploit the injectivity property of Haskell type constructors (usually not considered in TRS). We have therefore devised a novel completion algorithm that satisfies all our requirements; this algorithm is our main technical contribution.

Our completion algorithm comprises the following steps:

- TOP: E_g is normalised with respect to E_t , e.g. $\text{Int} \sim \text{F Bool}$ is normalised to $\text{Int} \sim \text{Char}$ with respect to $\text{F Bool} = \text{Int}$, exposing the inconsistency.
- TRIVIAL: Trivial equations are dropped, e.g. $\text{F a} \sim \text{F a}$, avoiding trivial non-termination.
- DECOMP: Non-essential type constructors are dropped, e.g. $(\text{F a}, \text{F b}) \sim (\text{Int}, \text{Bool})$ becomes $\text{F a} \sim \text{Int}$ and $\text{F b} \sim \text{Bool}$.
- SWAP: Equations are oriented properly, e.g. $\text{Int} \sim \text{F Bool}$ becomes $\text{F Bool} \sim \text{Int}$.
- SUBST: of E_g are substituted in each other, exposing inconsistencies. E.g. $\text{F a} \sim \text{Int}$ is substituted in $\text{F a} \sim \text{Char}$, resulting in $\text{Int} \sim \text{Char}$.

Moreover, our completion algorithm successfully deals with particularly difficult given equations like $\text{F Int} \sim [\text{G} (\text{F Int})]$. From left-to-right, the equation is non-terminating (the left-hand side occurs in the right-hand side), while the SWAP rule rejects the right-to-left orientation. Our solution is to break the equation into two new equations: $\text{F Int} \sim \mathbf{a}$ and $[\text{G} (\text{F Int})] \sim \mathbf{a}$ where \mathbf{a} is a skolem constant. After further completion we end up with $\text{F Int} \sim \mathbf{a}$ and $[\text{G} \mathbf{a}] \sim \mathbf{a}$ which is a proper strongly-normalising TRS.

An inconsistency discovered during completion, e.g. $\text{Int} \sim \text{Char}$, means that no evidence can be provided to support the given equations. While not ill-typed, the code under consideration is effectively unreachable.⁶

3.4 Restrictions on type function definitions

We already mentioned that a number of conditions must be imposed on the top-level type function definitions for reasons of soundness and completeness of our type checking strategy. The ground rules are these:

Modularity type instance declarations may be added one at a time, and must be individually accepted or rejected. It is not acceptable to require a global analysis of all the **type instance**, followed by a “yes” or “no” answer.

⁶ Our implementation raises an error to alert the programmer.

Arbitrary given constraints We may place restrictions on the `type instance` definitions, but we should place no restrictions on the additional given constraints E_g , because pattern matching on a GADT can give rise to arbitrary constraints.

Simplicity The simpler the rules, the better.

As an example of the need for arbitrary given constraints E_g consider the following program:

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

f :: Eq (F a) (G a) -> Int
f EQ = ...
```

where F and G are type functions In the right hand side of `f`, we have the given equation $F\ a \sim G\ a$, and clearly we could have given rise to an arbitrary such equation simply by choosing a different type signature for `f`.

Confluence The TRS-based type checking strategy requires that the top-level equations are confluent. For terminating rewrite systems, confluence is a decidable property: the test is based on the normalisation of critical pairs [13]. For reasons of modularity and simplicity⁷, we propose more restrictive properties:

- 1a. The heads of type function definitions do not contain (nested) type functions.
- 1b. The heads of type function definitions may not overlap.

The first of these is analogous to requiring that the patterns in an ordinary function definition use only variables and constructors, but not functions.

The second ensures that only one equation can match, and hence their order does not matter. Remember that, unlike Haskell function definitions, but like instance declarations, the `type instance` declarations for a type function are not required to occur all together, and hence are un-ordered. For example, the rule excludes this non-confluent overlap:

```
type instance F Int = Bool
type instance F Int = Char
```

but also excludes this set of confluent definitions:

```
type instance F Int = G Bool
type instance F Int = G Char

type instance G Bool = ()
type instance G Char = ()
```

⁷ from the points of view of programmers and compiler writers

Termination Next to confluence, termination of the TRS is essential for the completeness of our type checking strategy. The main principle for establishing termination in rule-based languages (to which type definitions belong) is that of decreasing calls [2]. A level-mapping assigns a value to all function calls; and all rules must satisfy the property that the level mappings of all calls in the right-hand side are smaller than the level-mapping of the rule-head. State-of-the-art termination analysers, e.g. [10], are capable of automatically inferring level-mappings in terms of various well-founded orders [7].

For reasons of modularity and simplicity, we propose not implement a state-of-the-art termination analysis, but rather to impose two simple conditions on all individual type definition clauses:

- 2a. The number of symbols (type constructors and schema variables) in each type function call in the body, is smaller than the number of the head.
- 2b. The number of occurrences of any schema variable in each type function call in the body, is smaller than the number of the head.

Completion Perhaps surprisingly, confluence and termination of the top-level equations is not enough. We must still account the completion of the given equations. Let's consider a single `type instance` that respects all the above conditions:

```
type instance H [[a]] = H (G a)
```

and the single given equation `G Int ~ [[Int]]`. The completion algorithm preserves this given equation, and yet the union of the two equations is not terminating: `H [[Int]] → H (G Int) → H [[Int]] → ...`

It turns out that the problem is caused by the nested function call `H (G a)`. We have gained much insight by expressing our problem as a set of Constraint Handling Rules (CHRs); in that setting, a nested function call corresponds to a “non-range restricted simplification rule”, which is known to be symptomatic of termination problems [22].

Our current solution is simple, if brutal; we add one further restriction:

3. No type function call may occur inside another type function call in a type definition clause.

Sadly, this restriction renders illegal a class of useful (usually closed) functions, e.g.:

```
type instance Mult Z      m = Z
type instance Mult (S n) m = Sum (Mult n m) m
```

Perhaps a more relaxed rule would suffice, a question we leave for future work.

3.5 Type-directed compilation

In a type-directed compiler, the type checker's task goes beyond providing a simple yes (the program is well-typed) or no (it's not). It must also generate the

necessary type information to enable the desugaring of the source language into the strongly-typed intermediate language. Hence, we adapt our type checking algorithm to generate type information for System F_C [23]. This is an extension of System F , which has been specifically designed as a practical compiler backed for Haskell, and is in actual use in GHC.

Encoding System F_C already has the essential ingredients, type functions and equality coercions, which have already proven their usefulness for encoding GADTs and associated type synonyms.

System F_C 's type functions and their definition are essentially identical to those in the source language, but the equality coercions deserve a little explanation. Thanks to its syntax-directedness, type checking in System F_C is much cheaper than in Haskell: declared and inferred types are checked for syntactic equivalence, e.g. `Int ≡ Int`.

However, the inference of non-syntactical equivalence proofs, like `F Int ~ Bool`, is problematic in System F_C . The reason is that the set of equational axioms in System F_C may be inconsistent. In particular, the internally consistent set of type function clauses may be at odds with the `newtype` axioms.

Example 1. The `newtype X = Int` is encoded in System F_C as an axiom `X ~ Int`, which conflicts with the type function:

```
type instance F X    = Char
type instance F Int = Bool
```

We can show that `F X` is both equal to `Char` and `Bool`, the former via the first clause of `F` and the latter via the `newtype` axiom and the second clause.

Fortunately, it is not necessary to repeat a proof that was already made by the Haskell type checker. The Haskell type checker can create a witness γ for the proof. Now the System F_C type checker can simply check the proof, based on its witness, rather than to infer it anew. This avoids the unsoundness pitfall and, as a bonus, checking a proof is also much cheaper than inferring it.

In System F_C , evidence is represented by a *coercion* γ , a special form of types whose kind is an equation, e.g. $\gamma : F \text{ Int} \sim \text{Bool}$ means that γ is evidence for the equation `F Int ~ Bool`. Coercion constants are denoted C and coercion variables co .

In System F_C , a unique coercion constant is associated with every type function clause, e.g. `C : type instance F Int = Bool`. Similarly, a given equational constraint, translates to a coercion variable, e.g.

```
id :: forall a b . a ~ b => a -> b
id = \x -> x
```

is encoded in System F_C as:

```
id :: forall a b . a ~ b => a -> b
id x = id =  $\Lambda(a:*) . \Lambda(b:*) . \Lambda(co:a \sim b) . \lambda(x:a) . (...)$ 
```

In order to type check an expression x whose inferred and expected types are a and b respectively, it has to be *cast* with the appropriate the evidence: $e \blacktriangleright \gamma$ where γ has kind $a \sim b$. Because types are eventually erased, these casts do not incur any runtime overhead.

Example 2. The full encoding of the above `id` function in System F_C is:

```
id =  $\Lambda(a:*) . \Lambda(b:*) . \Lambda(\text{co}:a \sim b) . \lambda(x:a) . (x \blacktriangleright \text{co})$ 
```

Complex coercions can be constructed from primitive coercions with coercion constructors:

- `sym` γ has kind $a \sim b$ if γ has kind $a \sim b$.
- $\gamma_1 \circ \gamma_2$ has kind $a \sim c$ if γ_1 has kind $a \sim b$ and γ_2 has kind $b \sim c$.
- $T \gamma$ has kind $T a \sim T b$ if γ has kind $a \sim b$, where T is a type constructor.
- $T \gamma$ has kind $T a \sim T b$ if γ has kind $a \sim b$, where T is a type constructor.
- `decompT,i` γ has kind $a_i \sim b_i$ if γ has kind $T \bar{a} \sim T \bar{b}$, where T is a type constructor.

Example 3. Using the previously defined type function `F`, the program:

```
main :: F Int
main = id True
```

is encoded in System F_C as:

```
main :: F Int
main = id @ Bool @ (F Int) @ (sym C) True
```

where type applications are denoted by `@`.

Coercion generation Given the appropriate coercions, the System F_C encoding of a Haskell program is a pretty straightforward matter. The hard part is of course the generation of these appropriate coercions, a task of the Haskell type checker.

Whenever the Haskell type checker constructs a wanted equation $\tau_1 \sim \tau_2$, i.e. to equate the inferred and expected types τ_1 and τ_2 of an expression e :

1. it creates a fresh *unknown* coercion γ of kind $\tau_1 \sim \tau_2$,
2. it inserts a cast in the code: $e \blacktriangleright \gamma$, and
3. it associates the coercion with the wanted equation, denoted $\gamma : \tau_1 \sim \tau_2$.

Whenever the type checker discharges a wanted equation, it fills in the unknown coercion, e.g. $\gamma := \gamma'$ where γ' has the same kind as γ . This is the *hard* part: how do we track the coercions of the top-level and given equations through the rewriting process of our type checking algorithm?

Firstly, it's not a simple matter of matching up some given equation with a *whole* wanted equation. Our algorithm is based on rewriting *individually* the left- and right-hand sides of a wanted equation, i.e. $\tau_1 \mapsto^* \tau$ and $\tau_2 \mapsto^* \tau$, to obtain a trivial equation of the form $\tau \sim \tau$.

Hence, we must construct two coercions γ_1 and γ_2 , one to justify each rewriting, i.e. $\gamma_1 : \tau_1 \sim \tau$ and $\gamma_2 : \tau_2 \sim \tau$. From these two coercions we can then determine the unknown coercion: $\gamma := \gamma_1 \circ \text{sym } \gamma_2$.

Example 4. Given these two clauses:

```
C1 : type instance F Int = Bool
C2 : type instance F Char = Bool
```

the type checker rewrites the left- and right-hand sides of the wanted equation $\gamma : F \text{ Int} \sim F \text{ Char}$ to Bool with coercions **C1** and **C2** respectively. Hence, the unknown coercion γ is determined as $C1 \circ \text{sym } C2$.

Secondly, we have to account for the completion phase. In the completion phase, the given equations are transformed. Hence, its corresponding evidence has to be transformed accordingly. For that purpose, all the steps in the completion algorithm of Section 3.3 have to be augmented with coercion transformations. For example:

- $\gamma : \text{Bool} \sim F \text{ Int}$ transformed with the SWAP step, becomes $\text{sym } \gamma : F \text{ Int} \sim \text{Bool}$,
- $\gamma : (F \text{ a}, F \text{ b}) \sim (\text{Int}, \text{Bool})$ decomposed with **Decomp**, becomes $\text{decomp}_{(\cdot),1} \gamma : F \text{ a} \sim \text{Int}$ and $\text{decomp}_{(\cdot),2} \gamma : F \text{ b} \sim \text{Bool}$.

4 Type functions versus functional dependencies

One of the most hotly debated questions in the latest standardisation process of the Haskell language (Haskell Prime [17]) is:

Should Haskell Prime adopt either functional dependencies or associated type synonyms?

There is little sense in providing two features for expressing functional relations. The above question now subsumed by a new one:

Should Haskell Prime adopt either functional dependencies or *type functions*?

We see three possible reasons for preferring type functions:

1. Type functions are inherently more familiar to functional programmers: it is a small step from functions at the value level to functions at the type level.
2. Type functions have their uses outside of type classes. Similar encodings with functional dependencies are rather bloated.
3. While functional dependencies have been around for quite a while now, it seems type checking for them is still rather ill-understood. In contrast, our prototype implementation of type functions type checks has no problems with GHC's open bugs related to functional dependencies.

However, before we consider the question from the point of view of language and compiler design (as the above arguments do), we first have to study a more pressing matter:

Is either of functional dependencies or type functions more expressive than the other?

While we do not yet have a formal result, we claim that both language features are indeed equally expressive. In the remainder of this section we justify our claim constructively and present translations both ways. Hence, language designers and compiler writers can happily disagree: the first group gets to choose what language feature to program in and the second group what language feature to implement.

4.1 From functional dependencies to type functions

We claim that every program involving functional dependencies can be re-expressed to one involving only type functions. This can often be done in an idiomatic way; for example, consider the way in which we re-expressed the two-parameter `Collects` class using a single-parameter class together with a type function (Section 2.1). But it is less clear how to translate classes with multiple or bi-directional functional dependencies, such as

```
class C a b | a -> b, b -> a where ..
```

Furthermore, if one starts with an existing program, the idiomatic translation is somewhat invasive because every occurrence of `Collects` must be changed to remove a type parameter, and new equality constraints must sometimes be added. For example,

```
merge :: (Collects c1 e, Collects c2 e) => c1 -> c2 -> c2
```

must become that of Section 2.4:

```
merge :: (Collects c1, Collects c2, Elem c1 ~ Elem c2) => c1 -> c2 -> c2
```

Thus motivated, we have developed an alternative, minimally invasive translation scheme from functional dependencies to type functions. The scheme is minimally invasive because it only affects `class` and `instance` declarations, and leaves all else untouched.

It works as follows. In the `class` declaration each functional dependency $\bar{a} \rightarrow b$ is replaced by: (1) a new (associated) type function $F \bar{a}$ and (2) a context constraint $F \bar{a} \sim b$. In every `instance` the proper type function instance is added.

Example 5. The transformed type class for collections is:

```
class Elem c ~ e => Collects c e where
  type Elem c
```

```
instance Collects [e] e where
  type Elem [e] = e
```

```
instance Collects BitSet Char where
  type Elem BitSet = Char
```

4.2 From type functions to functional dependencies

At first sight, the second part of the question should be answered negatively: as type functions do not have to be associated with type classes they are strictly more expressive. However, we can of course consider a fresh type class with functional dependencies (but no methods) to replace a stand-alone type family.

Example 6. The stand-alone type function `Sum` could be replaced by:

```
class Sum a b c | a b -> c

instance Sum Zero b b
instance Sum a b c => Sum (Succ a) b (Succ c)
```

In general, we can replace an n -ary type function with an $(n + 1)$ -ary type class, with a functional dependency from the n first arguments to the last one. Every function instance becomes a class instance where the n arguments of the LHS make up the n first arguments and the RHS becomes the $(n+1)$ th argument. Any function calls in the RHS have to be flattened into relational form in the instance context.

Matters become more complicated when data types are involved. For example, the following is perfectly legal in our system:

```
data T c = MkT [Elem c]
```

That is, a value of type `T c` is a `MkT` constructor wrapping a list of elements of collection type `c`. Notice that no type-class constraints are involved here. It is unclear how to translate this to functional dependencies. Certainly, we must add a new type parameter to the data type `T`, but then we need a way to express the connection between the two parameters. Something like this, perhaps?

```
data Collects c e => T c e = MkT [e]
```

But it is not clear that the `Collects c e` context on this declaration has the “right” effect. Similar complications arise with type synonyms and `newtypes`; see [8, Chapter 5]. A substantial advantage of our approach is that these complications go away.

5 Related work

Existing languages with type functions differ on various accounts from Haskell type functions. They only offer a fixed set of predefined functions (e.g. `ATS` [6]), type checking is incomplete (e.g. `Cayenne` [1], `Epigram` [15], `Omega` [21]) or the programmer has to construct the proofs himself (e.g. `LH` [14]). Moreover, all these languages assume that type functions are closed. More closely related to our work is the `Chameleon` system described in [25]. `Chameleon` makes use of the `Constraint Handling Rules (CHR)` [9] formalism for the specification of type class and type improvement relations. `CHR` is a committed-choice language

consisting of constraint rewrite rules. We expect to model open type functions via CHR rewrite rules which hopefully allows us to transfer some of the existing CHR type inference results [22] to the type function setting. The hard part so far has been modelling the treatment of evidence in CHR, which is reasonably straightforward in our current TRS formalism.

Both Neubauer et al. [16] and Diatchki [8] propose a functional notation for type classes with a functional dependencies. However, this notation is essentially syntactical sugar for the conventional relational notation of type classes. So these approaches gain the convenience of a functional notation, but miss the other advantages of our approach, especially concerning the use of type functions in the definition of data types.

6 Conclusion & future work

We have presented type functions, open functions at the type level. While they're equally expressive as functional dependencies when used with type classes, type functions can also be put to good use with GADTs and phantom types. We sketched a type checking strategy based on completion and term rewriting. Our implementation of type functions is available in the GHC HEAD branch, and is documented at http://haskell.org/haskellwiki/GHC/Type_families.

In future work we would like to extend the decidable class of type functions. It seems that closed type functions would allow us to relax a number of current modularity restrictions. Moreover, they should allow for additional proof strength. For example, we cannot currently show that `Sum a Zero ~ a` because `Sum` is an open function. Valid extensions of the function, like `Sum Int Zero = Bool`, do not satisfy this property.

The further comparison of functional dependencies and type functions is of great interest. We believe that type functions are a better choice, from the point of view of both language design and type checking.

Finally, the performance of the our type checking algorithm deserves further attention. In particular we should establish its worst and average time complexities, and the impact on programs that do not involve type functions.

Acknowledgments

We would like to thank Roman Leshchinskiy for his comments and for uprooting bugs in our preliminary implementation, and James Chapman for explaining the current state of Epigram.

Part of this work was conducted during an internship of Tom Schrijvers at Microsoft Research Cambridge.

References

1. L. Augustsson. Cayenne - a language with dependent types. In *Proc. of ICFP'98*, pages 239–250. ACM Press, 1998.

2. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
3. W. Boone. The word problem. *Annals of Mathematics*, 70:207–265, 1959.
4. M. Chakravarty, G. Keller, S. Peyton Jones, and S. Marlow. Associated types with class. In *ACM Symposium on Principles of Programming Languages (POPL'05)*. ACM, 2005.
5. M. M. T. Chakravarty, G. Keller, and S. P. 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. C. Chen and H. Xi. Combining programming with theorem proving. In *Proc. of ICFP'05*, pages 66–77. ACM Press, 2005.
7. S. Decorte, D. D. Schreye, and M. Fabris. Automatic inference of norms: a missing link in automatic termination analysis. In *ILPS '93: Proceedings of the 1993 international symposium on Logic programming*, pages 420–436, Cambridge, MA, USA, 1993. MIT Press.
8. I. S. Diatchki. *High-level abstractions for low-level programming*. PhD thesis, OGI School of Science & Engineering, May 2007.
9. T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1–3):95–138, 1998.
10. J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Automated Termination Proofs with AProVE. In *Proceedings of the 15th International Conference on Rewriting Techniques and Applications (RTA-04)*, volume 3091 of *Lecture Notes in Computer Science*, pages 210–220, Aachen, Germany, 2004.
11. R. Hinze. Fun with phantom types. In J. Gibbons and O. de Moor, editors, *The Fun of Programming*, pages 245–262. Palgrave Macmillan, 2003. ISBN 1-4039-0772-2 hardback, ISBN 0-333-99285-7 paperback.
12. 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.
13. D. Knuth and P. Bendix. Simple word problems in universal algebras. *Computational Problems in Abstract Algebra*, pages 263–297, 1970.
14. D. R. Licata and R. Harper. A formulation of Dependent ML with explicit equality proofs. Technical Report CMU-CS-05-178, Carnegie Mellon University Department of Computer Science, 2005.
15. C. McBride. Epigram: A dependently typed functional programming language. <http://www.dur.ac.uk/CARG/epigram/>.
16. M. Neubauer, P. Thiemann, M. Gasbichler, and M. Sperber. A functional notation for functional dependencies. In *Proceedings of the 2001 Haskell Workshop*, 2001.
17. S. Peyton-Jones et al. The Haskell Prime Report, 2007. Working draft.
18. S. Peyton Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17:1–82, Jan. 2007.
19. D. A. Plaisted and A. Sattler-Klein. Proof lengths for equational completion. *Inf. Comput.*, 125(2):154–170, 1996.
20. T. Schrijvers, M. Sulzmann, S. Peyton-Jones, and M. Chakravarty. Type checking for type functions. Draft report available from the authors, July 2007.
21. T. Sheard. Type-level computation using narrowing in Omega. In *Proceedings of the Programming Languages meets Program Verification (PLPV 2006)*, volume 174 of *Electronic Notes in Computer Science*, pages 105–128, 2006.
22. P. J. Stuckey and M. Sulzmann. A theory of overloading. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(6):1–54, 2005.

23. M. Sulzmann, M. M. T. Chakravarty, S. P. Jones, and K. Donnelly. System F with type equality coercions. In *TLDI '07: Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 53–66, New York, NY, USA, 2007. ACM Press.
24. M. Sulzmann, T. Schrijvers, and P. J. Stuckey. Type inference for GADTs via Herbrand constraint abduction. Manuscript, July 2006.
25. 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.