Fun with type functions

Oleg Kiselyov   Simon Peyton Jones   Chung-chieh Shan

May 3, 2010

Abstract

Tony Hoare has always been a leader in writing down and proving properties of programs. To prove properties of programs automatically, the most widely used technology today is by far the ubiquitous type checker. Alas, static type systems inevitably exclude some good programs and allow some bad ones. Thus motivated, we describe some fun we have been having with Haskell, by making the type system more expressive without losing the benefits of automatic proof and compact expression. Specifically, we offer a programmer's tour of so-called *type families*, a recent extension to Haskell that allows functions on types to be expressed as straightforwardly as functions on values. This facility makes it easier for programmers to effectively extend the compiler by writing functional programs that execute during type-checking.

This paper gives a programmer's tour of type families as they are

supported in GHC.

Source code for all the examples is available at `http://research.microsoft.com/~simonpj/papers/assoc-types/fun-with-type-funs/fun-with-type-funs.zip`

1 Introduction

The type of a function specifies (partially) what it does. Although weak as a specification language, static types have compensating virtues: they are

- *lightweight*, so programmers use them;
- *machine-checked* with minimal programmer assistance;
- *ubiquitous*, so programmers cannot avoid them.

As a result, static type checking is by far the most widely used verification technology today.

Every type system excludes some “good” programs, and permits some “bad” ones. For example, a language that lacks polymorphism will reject this “good” program:

```
f :: [Int] -> [Bool] -> Int
f is bs = length is + length bs
```

Why? Because the `length` function cannot apply to both a list of `Int`s and a list of `Bool`s. The solution is to use a more sophisticated type system in which we can give `length` a polymorphic type.

Conversely, most languages will accept the expression

```
speed + distance
```

where `speed` is a variable representing speed, and `distance` represents distance, even though adding a speed to a distance is as much nonsense as adding a character to a boolean.

The type-system designer wants to accommodate more good programs and exclude more bad ones, without going overboard and losing the virtues mentioned above. In this paper we describe *type families*, an experimental addition to Haskell with precisely this goal. We start by using type families to accommodate more good programs, then turn in Section 5 to excluding more bad programs. We focus on the programmer, and our style is informal and tutorial. The technical background can be found elsewhere [5–7, 42]. The complete code described in the paper is available . That directory also contains the online version of the paper with additional appendices, briefly describing the syntax of type functions and the rules and pitfalls of their use. Appendix C gives an alternative derivation of typed `sprintf` using higher-order type-level functions.

2 Associated types: indexing types by

types

Haskell has long offered two ways to express *relations* on types. Multiparameter type classes express arbitrary, many-to-many relations, whereas type constructors express specifically *functional* relations, where one type (the 'argument') uniquely determines the other. For example, the relation between the type of a list and the type of that list's elements is a functional relation, expressed by the type constructor `[] :: * -> *`, which maps an arbitrary type `a` to the type `[a]` of lists of `a`. A type constructor maps its argument types *uniformly*, incorporating them into a more complex type without inspecting them. Type functions, the topic of this paper, also establish functional relations between types, but a type function may perform case analysis on its argument types.

For example, consider the relation between a monad that supports mutable state and the corresponding type constructor for reference cells. The `IO` monad supports the following operations on reference cells of type `IORef a`:

```
newIORef   :: a -> IO (IORef a)
readIORef  :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
```

Similarly, the `ST s` monad supports the analogous operations on reference cells of type `STRef s a`:

```
newSTRef   :: a -> ST s (STRef s a)
readSTRef  :: STRef s a -> ST s a
```

```
writeSTRef :: STRef s a -> a -> ST s ()
```

2

It is tempting to overload these operations using a multiparameter type class:

```
class Mutation m r where
  newRef   :: a -> m (r a)
  readRef  :: r a -> m a
  writeRef :: r a -> a -> m ()

instance Mutation IO IORef where
  newRef = newIORef
  ...etc...

instance Mutation (ST s) (STRef s) where
  newRef = newSTRef
  ...etc...
```

This approach has two related disadvantages. First, the types of `newRef` and the other class operations are too polymorphic: one could declare an instance such as

```
instance Mutation IO (STRef s) where ...
```

even though we intend that the `IO` monad has *exactly one* reference type, namely `IORef`. Second, as a result, it is extremely easy to write programs with ambiguous typings, such as

```
readAndPrint :: IO ()
readAndPrint = do { r <- newRef 'x'; v <- readRef r; print v }
```

We know, from the type signature, that the computation is performed in the `IO` monad, but type checker cannot select the type of `r`, since the `IO` monad could have reference cells of many different types. Therefore, we must annotate `r` with its type explicitly. Types are no longer lightweight when they have to be explicitly specified even for such a simple function.

The standard solution to the second problem is to use a *functional dependency*:

```
class Mutation m r | m -> r where ...
```

The "`m -> r`" part says that every `m` is related to at most one `r`. Functional dependencies have become a much-used extension of Haskell, and we return to a brief comparison in Section 6. Meanwhile, the main purpose of this paper is to explain an alternative approach in which we express the functional dependency at the type level in an explicitly functional way.

2.1 Declaring an associated type

The class `Mutation` does not *really* have two type parameters: it has one type parameter, associated with another type that is functionally dependent. Type families allow one to say this directly:

```
class Mutation m where
  type Ref m :: * -> *
```

```haskell
  newRef   :: a -> m (Ref m a)
  readRef  :: Ref m a -> m a
  writeRef :: Ref m a -> a -> m ()
```



```haskell
instance Mutation IO where
  type Ref IO = IORef
  newRef   = newIORef
  readRef  = readIORef
  writeRef = writeIORef

instance Mutation (ST s) where
  type Ref (ST s) = STRef s
  newRef   = newSTRef
  readRef  = readSTRef
  writeRef = writeSTRef
```

The `class` declaration now introduces a *type function* `Ref` (with a specified kind) alongside the usual *value functions* such as `newRef` (each with a specified type). Similarly, each `instance` declaration contributes a clause defining the type function at the instance type alongside a witness for each value function.

We say that `Ref` is a *type family*, or an *associated type* of the class `Mutation`. It behaves like a function at the type level, so we also call `Ref` a *type function*. Applying a type function uses the same syntax as applying a type constructor: `Ref m a` above means to apply the type function `Ref` to `m`, then apply the resulting type constructor to `a`.

The types of `newRef` and `readRef` are now more perspicuous:

```
newRef  :: Mutation m => a -> m (Ref m a)
readRef :: Mutation m => Ref m a -> m a
```

Furthermore, by omitting the functionally determined type parameter from `Mutation`, we avoid the ambiguity problem exemplified by `readAndPrint` above. When performing type inference for `readAndPrint`, the type of `r` is readily inferred to be `Ref IO Char`, which the type checker reduces to `IORef Char`. In general, the type checker reduces `Ref IO` to `IORef`, and `Ref (ST s)` to `STRef s`.

These type equalities aside, `Ref` behaves like any other type constructor, and it may be used freely in type signatures and data type declarations. For example, this declaration is fine:

```
data T m a = MkT [Ref m a]
```

2.2 Arithmetic

In the class `Mutation` of Section 2.1, we used an associated type to avoid a two-parameter type class, but that is not to say that associated types obviate multiparameter type classes. By declaring associated types in multiparameter type classes, we introduce type functions that take multiple arguments. One compelling use of such type functions is to make type coercions implicit, especially in arithmetic. Suppose we want to be able to write `add a b` to add two numeric values `a` and `b` even if one is an `Integer` and the other is a `Double` (without writing `fromIntegral` explicitly). We also want to add a scalar to a vector represented by a list without writing `repeat` explicitly to coerce the scalar to the vector type. The result type should be the simplest that is compatible with both operands. We can express this intent using a two-parameter type class, whose parameters

are the argument types of `add`, and whose associated type `SumTy` is the result:

```
class Add a b where
  type SumTy a b
  add :: a -> b -> SumTy a b

instance Add Integer Double where
  type SumTy Integer Double = Double
  add x y = fromIntegral x + y

instance Add Double Integer where
  type SumTy Double Integer = Double
  add x y = x + fromIntegral y

instance (Num a) => Add a a where
  type SumTy a a = a
  add x y = x + y
```

In other words, `SumTy` is a two-argument type function that maps the argument types of an addition to type of its result. The three instance declarations explain how `SumTy` behaves on arguments of various types. We can then write `add (3::Integer) (4::Double)` to get a result of type `SumTy Integer Double`, which is the same as `Double`.

The same technique lets us conveniently build homogeneous lists out of heterogeneous but compatible components:

```
class Cons a b where
  type ResTy a b
  cons :: a -> [b] -> [ResTy a b]

instance Cons Integer Double where
  type ResTy Integer Double = Double
  cons x ys = fromIntegral x : ys

-- ...
```

With instances of this class similar to those of the class `Add`, we can `cons` an `Integer` to a list of `Doubles` without any explicit conversion.

2.3 Graphs

Garcia et al. [15] compare the support for generic programming offered by Haskell, ML, C++, C#, and Java. They give a table of qualitative conclusions, in which Haskell is rated favourably in all respects except associated types. This observation was one of the motivations for the work we describe here. Now that GHC supports type functions, we can express their main example, but we need an additional kind of associated type, introduced with the `data` keyword, which is described in Section 2.4 below:

```
class Graph g where
  type Vertex g
  data Edge g
  src, tgt :: Edge g -> Vertex g
  outEdges :: g -> Vertex g -> [Edge g]
```



```
newtype G1 = G1 [Edge G1]
instance Graph G1 where
  type Vertex G1 = Int
  data Edge   G1 = MkEdge1 (Vertex G1) (Vertex G1)
  -- ...definitions for methods...

newtype G2 = G2 (Map (Vertex G2) [Vertex G2])
instance Graph G2 where
  type Vertex G2 = String
  data Edge   G2 = MkEdge2 Int (Vertex G2) (Vertex G2)
  -- ...definitions for methods...
```

The class `Graph` has two associated types, `Vertex` and `Edge`. We show two representative instances. In `G1`, a graph is represented by a list of its edges, and a vertex is represented by an `Int`. In `G2`, a graph is represented by a mapping from each vertex to a list of its immediate neighbours, a vertex is represented by a `String`, and an `Edge` stores a weight (of type `Int`) as well as its end-points. As these instance declarations illustrate, the declaration of a `Graph` instance is free to use the type functions `Edge` and `Vertex`.

2.4 Associated data types

The alert reader will notice in the class `Graph` that the associated type for `Edge` is declared using "`data`" rather than "`type`". Correspondingly, the `instance` declarations give a `data` declaration for `Edge`, complete with data constructors `MkEdge1` and `MkEdge2`. The reason for this use of `data` is somewhat subtle.

A type constructor such as `[]` expresses a functional relation between types that is *injective*, mapping different argument types to different results. For example, if two list types are the same, then their element types must be the same, too. This injectivity does not generally hold for type functions. Consider this function to find the list of vertices adjacent to the given vertex `v` in the graph `g`:

```
neighbours g v = map tgt (outEdges g v)
```

We expect GHC to infer the following type:

```
neighbours :: Graph g => g -> Vertex g -> [Vertex g]
```

Certainly, `outEdges` returns a `[Edge g1]` (for some type `g1`), and `tgt` requires its argument to be of type `Edge g2` (for some type `g2`). So, GHC's type checker requires that `Edge g1` ∼ `Edge g2`, where "∼" means type equality.[1] Does that mean that `g1` ∼ `g2`, as intuition might suggest? Not necessarily! If `Edge` were an associated **type**, rather than **data**, we could have written these instances:

```
instance Graph G3 where
  type Edge G3 = (Int,Int)
instance Graph G4 where
  type Edge G4 = (Int,Int)
```

[1] "=" is used for too many other things.

so that `Edge G3` ∼ `Edge G4` even though `G3` and `G4` are distinct. In that case, the inferred type of `neighbours` would be:

```
neighbours :: (Graph g1, Graph g2, Edge g1 ~ Edge g2)
           => g1 -> Vertex g1 -> [Vertex g2]
```

Although correct, this type is more general and complex than we want. By declaring `Edge` with `data`, we specify that `Edge` is injective, that `Edge g1` ∼ `Edge g2` indeed implies `g1` ∼ `g2`.[2] GHC then infers the simpler type we want.

2.5 Type functions are open

Value-level functions are *closed* in the sense that they must be defined all in one place. For example, if one defines

```
length :: [a] -> Int
```

then one must give the *complete* definition of `length` in a single place:

```
length []     = 0
length (x:xs) = 1 + length xs
```

It is not legal to put the two equations in different modules.

In contrast, a key property of type functions is that, like type classes themselves, they are *open* and can be extended with additional instances at any time. For example, if next week we define a new type `Age`, we can extend `SumTy` and `add` to work over `Age` by adding an instance declaration:

```
newtype Age = MkAge Int

instance Add Age Int where
  type SumTy Age Int = Age
  add (MkAge a) n = MkAge (a+n)
```

We thus can add an `Int` to an `Age`, but not an `Age` or `Float` to an `Age` without another instance.

2.6 Type functions may be recursive

Just as the instance for `Show [a]` is defined in terms of `Show a`, a type function is often defined by structural recursion on the input type. Here is an example, extending our earlier `Add` class with a new instance:

```
instance (Add Integer a) => Add Integer [a] where
  type SumTy Integer [a] = [SumTy Integer a]
  add x y = map (add x) y
```

Thus

`SumTy Integer` [`Double`] $\sim$ [`SumTy Integer Double`] $\sim$ [`Double`].

In a similar way, we may extend the `Mutation` example of Section 2.1 to *monad transformers*. Recall that a monad transformer `t :: (*->*) -> (*->*)`

[2] A possible extension, not currently implemented by GHC, would be to allow an associated `type` synonym declaration optionally to specify that it should be injective, and to check that this property is maintained as each `instance` is added.

is a higher-order type constructor that takes a monad `m` into another monad `t m`.

```
class MonadTrans t where
  lift :: Monad m => m a -> t m a
```

At the value level, `lift` turns a monadic computation (of type `m a`) into one in the transformed monad (of type `t m a`). Now, if a monad `m` is an instance of `Mutation`, then we can make the transformed monad `t m` into such an instance too:

```
instance (Monad m, Mutation m, MonadTrans t)
```

```
    => Mutation (t m) where
  type Ref (t m) = Ref m
  newRef   = lift . newRef
  readRef  = lift . readRef
  writeRef = (lift .) . writeRef
```

The equation for `Ref` says that the type of references in the transformed monad is the same as that in the base monad.

3 Optimised container representations

A common optimisation technique is to represent data of different types differently (rather than uniformly as character strings, for example). This technique is best known when applied to container data. For example, we can use the same array container to define a `Bool` array and to define an `Int` array, yet a `Bool` array can be stored more compactly and negated elementwise faster when its elements are tightly packed as a bit vector. C++ programmers use template meta-programming to exploit this idea to great effect, for example in the Boost library [47]. The following examples show how to express the same idea in Haskell, using type functions to map among the various concrete types that represent the same abstract containers.

3.1 Type-directed memoization

To memoise a function is to improve its future performance by recording and reusing its past behaviour in a *memo table* [35]. The memo table augments the concrete representation of the function without affecting its abstract interface. A typical way to implement memoization is to add a lookup from the table on entry to the function and an update to the table on exit from the function. Haskell offers an elegant way to express memoization, because we can use lazy evaluation to manage the lookup and update of the memo table. But type functions offer a new possibility: *the type of the memo table can be determined automatically from the argument type of the memoised function* [12, 19].

We begin by defining a type class `Memo`. The constraint `Memo a` means that the behaviour of a function from an argument type `a` to a result type `w` can be represented as a memo table of type `Table a w`, where `Table` is a type function that maps a type to a constructor.


```
class Memo a where
  data Table a :: * -> *
  toTable :: (a -> w) -> Table a w
  fromTable :: Table a w -> (a -> w)
```

For example, we can memoise any function from `Bool` by storing its two return values as a lazy pair. This lazy pair is the memo table.

```
instance Memo Bool where
  data Table Bool w = TBool w w
  toTable f = TBool (f True) (f False)
  fromTable (TBool x y) b = if b then x else y
```

To memoise a function `f :: Bool -> Int`, we simply replace it by `g`:

```
g :: Bool -> Int
g = fromTable (toTable f)
```

The first time `g` is applied to `True`, the Haskell implementation computes the first component of the lazy pair (by applying `f` in turn to `True`) and remembers it for future reuse. Thus, if `f` is defined by

```
f True  = factorial 100
f False = fibonacci 100
```

then evaluating `(g True + g True)` will take barely half as much time as evaluating `(f True + f True)`.

Generalising the `Memo` instance for `Bool` above, we can memoise functions from any sum type, such as the standard Haskell type `Either`:

```
data Either a b = Left a | Right b
```

We can memoise a function from `Either a b` by storing a lazy pair of a memo table from `a` and a memo table from `b`. That is, we take advantage of the isomorphism between the function type `Either a b -> w` and the product type `(a -> w, b -> w)`.

```
instance (Memo a, Memo b) => Memo (Either a b) where
  data Table (Either a b) w = TSum (Table a w) (Table b w)
  toTable f = TSum (toTable (f . Left)) (toTable (f . Right))
  fromTable (TSum t _) (Left  v) = fromTable t v
  fromTable (TSum _ t) (Right v) = fromTable t v
```

Of course, we need to memoise functions from `a` and functions from `b`; hence the "`(Memo a, Memo b) =>`" part of the declaration. Dually, we can memoise functions from the product type `(a,b)` by storing a memo table from `a` whose entries are memo tables from `b`. That is, we take advantage of the currying isomorphism between the function types `(a,b) -> w` and `a -> b -> w`.

```
instance (Memo a, Memo b) => Memo (a,b) where
  newtype Table (a,b) w = TProduct (Table a (Table b w))
  toTable f = TProduct (toTable (\x -> toTable (\y -> f (x,y))))
  fromTable (TProduct t) (x,y) = fromTable (fromTable t x) y
```


3.2 Memoisation for recursive types

What about functions from recursive types, like lists? No problem! A list is a combination of a sum, a product, and recursion:

```
instance (Memo a) => Memo [a] where
```

```
data Table [a] w = TList w (Table a (Table [a] w))
toTable f = TList (f [])
                  (toTable (\x -> toTable (\xs -> f (x:xs))))
fromTable (TList t _) []     = t
fromTable (TList _ t) (x:xs) = fromTable (fromTable t x) xs
```

As in Section 3.1, the type function `Table` is recursive. Since a list is either empty or not, `Table [Bool] w` is represented by a pair (built with the data constructor `TList`), whose first component is the result of applying the memoised function `f` to the empty list, and whose second component memoises applying `f` to non-empty lists. A non-empty list `(x:xs)` belongs to a product type, so the corresponding table maps each `x` to a table that deals with `xs`. We merely combine the memoization of functions from sums and from products.

It is remarkable how laziness takes care of the recursion in the type `[a]`. A memo table for a function `f` maps *every possible argument* `x` of `f` to a result `(f x)`. When the argument type is finite, such as `Bool` or `(Bool,Bool)`, the memo table is finite as well, but what if the argument type is infinite, such as `[Bool]`? Then, of course, the memo table is infinite: in the instance declaration above, we define `toTable` for `[a]` not only using `toTable` for `a` but also using `toTable` for `[a]` recursively. Just as each value `(f x)` in a memo table is evaluated only if the function is ever applied to that particular `x`, so each sub-table in this memo table is expanded only if the function is ever applied to a list with that prefix. So the laziness works at two distinct levels.

Now that we have dealt with sums, products, and recursion, we can deal with any data type at all. Even base types like `Int` or `Integer` can be handled by first converting them (say) to a list of digits, say `[Bool]`. Alternatively, it is equally easy to give a specialised instance for `Table Integer` that uses some custom (but infinite!) tree representation for `Integer`.

More generally, if we define `Memo` instances – once and for all – for sum types, product types, and fixpoint types, then we can define a `Memo` instance for some new type just by writing an isomorphism between the new type and a construction out of sum types, product types, and fixpoint types. These boilerplate `Memo` instances can in fact be defined generically, with the help of functional dependencies [8] or type functions.[3]

3.3 Generic finite maps

A *finite map* is a partial function from a domain of *keys* to a range of *values*. Finite maps can be represented using many standard data structures, such as binary trees and hash tables, that work uniformly across all key types. However, our memo-table development suggests another possibility, that of representing a finite map using a memo table:

[3] `http://hackage.haskell.org/cgi-bin/hackage-scripts/package/pointless-haskell`


```
type Map key val = Table key (Maybe val)
```

That is, we represent a *partial* function from `key` to `val` as a *total* function from `key` to `Maybe val`. But we get two problems. The smaller one is that whereas `Table` did not need an `insert` method – once we construct the memo table, we never need to update it – `Map` needs `insert` and many other methods including `delete` and `union`. These considerations might lead us to add `insert`, `delete`, etc. to the `Table` interface, where they appear quite out of place. A nicer alternative would be to define a subclass of `Table`.

The second, more substantial problem is that `Table` is unnecessarily inefficient in the way it represents keys that map to `Nothing`. An extreme case is an *empty* map whose key type is `Integer`. An efficient finite map would represent an empty map as an empty trie, so that the `lookup` operation returns immediately with `Nothing`. If instead we represent the empty map as an (infinite) `Table` mapping every `Integer` to `Nothing`, each lookup will explore a finite path in the potentially infinite tree, taking time proportional the number of bits in the `Integer`. Furthermore, looking up many `Integer`s in such a `Table` would force many branches of the `Table`, producing a large tree in memory, with `Nothing` in every leaf! Philosophically, it seems nicer to distinguish the mapping of a key to `Nothing` from the *absence* of the mapping for that key.

For these reasons, it makes sense to implement Map afresh [19, 22]. As with `Memo`, we define a class `Key` and an associated data type `Map`:

```
class Key k where
  data Map k :: * -> *
  empty  :: Map k v
```

```
  lookup :: k -> Map k v -> Maybe v
  -- ...many other methods could be added...
```

Now the instances follow in just the same way as before:

```
instance Key Bool where
  data Map Bool elt = MB (Maybe elt) (Maybe elt)
  empty = MB Nothing Nothing
  lookup False (MB mf _) = mf
  lookup True  (MB _ mt) = mt

instance (Key a, Key b) => Key (Either a b) where
  data Map (Either a b) elt = MS (Map a elt) (Map b elt)
  empty = MS empty empty
  lookup (Left  k) (MS m _) = lookup k m
  lookup (Right k) (MS _ m) = lookup k m

instance (Key a, Key b) => Key (a,b) where
  data Map (a,b) elt = MP (Map a (Map b elt))
  empty = MP empty
  lookup (a,b) (MP m) = case lookup a m of
                          Nothing -> Nothing
                          Just m' -> lookup b m'
```

The fact that this is a *finite* map makes the instance for `Int` easier than before, because we can simply invoke an existing data structure (a Patricia tree, for example) for finite maps keyed by `Int`:

```
instance Key Int where
  newtype Map Int elt = MI (Data.IntMap.IntMap elt)
  empty = MI Data.IntMap.empty
  lookup k (MI m) = Data.IntMap.lookup k m
```

Implementations of other methods (such as `insert` and `union`) and instances at other types (such as lists) are left as exercises for the reader.

Hutton describes another example with the same flavour [24].

3.4 Session types and their duality

We have seen a recursively defined correspondence between the type of keys and the type of a finite map over those keys. The key and the lookup function of a finite map can be regarded as a pair of *processes* that communicate in a particular way: the key sends indices to the lookup, then the lookup responds with the element's value. In this section, we generalise this correspondence to the relationship between a pair of processes that communicate with each other by sending and receiving values in a *session*.

For example, consider the following definitions:

```
data Stop = Done
newtype In  a b = In (a -> IO b)
data    Out a b = Out a   (IO b)
```

```
add_server :: In Int (In Int (Out Int Stop))
add_server = In $ \x -> return $ In $ \y ->
             do { putStrLn "Thinking"
                ; return $ Out (x + y) (return Done) }
```

The function-like value `add_server` accepts two `Int`s in succession, then prints "Thinking" before responding with an `Int`, their sum. We call `add_server` a *process*, whose interface protocol is specified by its type – so called *session type*. We write session types explicitly in this section, but they can all be inferred.

We may couple two processes whose protocols are complementary, or *dual*:

```
class Session a where
  type Dual a
  run :: a -> Dual a -> IO ()
```

Of course, to write down the definition of `run` we must also say what it means to be dual. Doing so is straightforward:

```
instance (Session b) => Session (In a b) where
  type Dual (In a b) = Out a (Dual b)
  run (In f) (Out a d) = f a >>= \b -> d >>= \c -> run b c

instance (Session b) => Session (Out a b) where
  type Dual (Out a b) = In a (Dual b)
  run (Out a d) (In f) = f a >>= \b -> d >>= \c -> run c b
```

```
instance Session Stop where
  type Dual Stop = Stop
  run Done Done = return ()
```


The type system guarantees that the protocols of the two processes match. Thus, if we write a suitable client `add_client`, like

```
add_client :: Out Int (Out Int (In Int Stop))
add_client = Out 3 $ return $ Out 4 $
             do { putStrLn "Waiting"
                ; return $ In $ \z -> print z >> return Done }
```

we may couple them (either way around):

```
> run add_server add_client
Thinking
Waiting
7
> run add_client add_server
Thinking
Waiting
7
```

However, `run` will not allow us to couple two processes that do not have dual protocols. Suppose that we write a negation server:

```
neg_server :: In Int (Out Int Stop)
neg_server = In $ \x ->
             do { putStrLn "Thinking"
                ; return $ Out (-x) (return Done) }
```

Then `(run add_client neg_server)` will fail with a type error. Just as the `Memo` class represents functions of type `a -> w` by memo tables of the matching type `Table a w`, this `Session` class represents consumers of type `a -> IO ()` by producers of the matching type `Dual a`.

These protocols do not allow past communication to affect the type and direction of *future* exchanges. For example, it seems impossible to write a well-typed server that begins by receiving a `Bool`, then performs addition if `True` is received and negation if `False` is received. However, we can express a protocol that chooses between addition and negation (or more generally, a protocol that chooses among a finite number of ways to continue). We simply treat such a binary choice as a distinct sort of protocol step. The receiver of the choice has a product type, whereas the sender has a sum type:

```
instance (Session a, Session b) => Session (Either a b) where
  type Dual (Either a b) = (Dual a, Dual b)
  run (Left  y) (x,_) = run y x
  run (Right y) (_,x) = run y x

instance (Session a, Session b) => Session (a, b) where
  type Dual (a,b) = Either (Dual a) (Dual b)
```

```
run (x,_) (Left  y) = run x y
run (_,x) (Right y) = run x y
```

These additional instances let us define a combined addition-negation server, along with a client that chooses to add. The two new processes sport (inferable) types that reflect their initial choice.


```
server :: (In Int (Out Int Stop),
           In Int (In Int (Out Int Stop)))
server = (neg_server, add_server)

client :: Either (Out Int (In Int Stop))
                 (Out Int (Out Int (In Int Stop)))
client = Right add_client
```

To connect `server` and `client`, we can evaluate either `run server client` or `run client server`. The session type of the client hides which of the two choices the client eventually selects; the choice may depend on user input at run time, which the type checker has no way of knowing. The type checker does statically verify that the corresponding server can handle either choice.

With the instances defined above, each protocol allows only a finite number of exchanges, so a server cannot keep looping until the client disconnects. This restriction is not fundamental: recursion in protocols can be expressed, for example using an explicit fixpoint operator at the type level [38].

We can also separate the notion of a *process* from that of a *channel*, and associate a protocol with the channel rather than the process. This and other variants have been explored in other works [26, 27, 36, 38, 41], from which we draw the ideas of this section in a simplified form.

In principle, we can require that `Dual` be an involution (that is, `Dual` be its own inverse) by adding a *equality constraint* as a superclass of `Session`:

```
class (Dual (Dual a) ~ a) => Session a where ...
```

We can then invoke `run` on a pair of processes without worrying about which process is known to be the dual of which other process. More generally, this technique lets us express bijections between types. However, such equality superclasses are not yet implemented in the latest release of GHC (6.10).

4 Typed sprintf and sscanf

We conclude the first half of the paper, about using type functions to accommodate more good programs, with a larger example: typed `sprintf` and `sscanf`.

A hoary chestnut for typed languages is the definition of `sprintf` and `sscanf`. Although these handy functions are present in many languages (such as C and Haskell), they are usually not type-safe: the type checker does not stop the programmer from passing to `sprintf` more or fewer arguments than required by the format descriptor. The typing puzzle is that we want the following to be true:

```
sprintf "Name=%s"          :: String -> String
sprintf "Age=%d"           :: Int -> String
sprintf "Name=%s, Age=%d" :: String -> Int -> String
```

That is, the *type* of `(sprintf fs)` depends on the *value* of the *format descriptor* `fs`. Supporting such dependency directly requires a full-spectrum

dependently typed language, but there is a small literature of neat techniques for getting close without such a language [1, 9, 20]. Here we show one technique using type families. In fact, we accomplish something more general: typed `sprintf` and `sscanf` *sharing the same format descriptor*. Typed `sprintf` has received a lot more attention than typed `sscanf`, and it is especially rare for an implementation of both to use the same format descriptor.

4.1 Typed sprintf

We begin with two observations:

- Format descriptors in C are just strings, which leaves the door wide open for malformed descriptors that `sprintf` does not recognise (e.g., `sprintf "%?"`). The language of format descriptors is a small domain-specific language, and the type checker should reject ill-formed descriptors.

- In Haskell, we cannot make the type of `(sprintf f)` depend on the value of the format descriptor `f`. However, using type functions, we can make it depend on the *type* of `f`.

Putting these two observations together suggests that we use a now-standard design pattern: a domain-specific language expressed using a generalised algebraic data type (GADT) indexed by a type argument. Concretely, we can define the type of format descriptors `F` as follows:

```
data F f where
  Lit :: String -> F L
  Val :: Parser val -> Printer val -> F (V val)
  Cmp :: F f1 -> F f2 -> F (C f1 f2)

data L
data V val
data C f1 f2

type Parser  a = String -> [(a,String)]
type Printer a = a -> String
```

So `F` is a GADT with three constructors, `Lit`, `Val`, and `Cmp`.[4] Our intention is that `(sprintf f)` should behave as follows:

- If `f` = `Lit s`, then print (that is, return as the output string) `s`.

- If `f` = `Cmp f1 f2`, then print according to descriptor `f1` and continue according to descriptor `f2`.
- If `f` = `Val r p`, then use the printer `p` to convert the first argument to a string to print. (The `r` argument is used for parsing in Section 4.2 below.)

If `fmt :: F ty`, then the *type* `ty` encodes the *shape* of the term `fmt`. For example, given `int :: F (V Int)`, we may write the following format descriptors:

[4] "`Cmp`" is short for "compose".


```
f_ld  = Lit "day"                               :: F L
f_lds = Cmp (Lit "day") (Lit "s")               :: F (C L L)
f_dn  = Cmp (Lit "day ") int                    :: F (C L (V Int))
f_nds = Cmp int (Cmp (Lit " day") (Lit "s")) :: F (C (V Int) (C L L))
```

In each case, the type encodes an abstraction of the value. (We have specified the types explicitly, but they can be inferred.) The types `L`, `V`, and `C` are type-level abstractions of the terms `Lit`, `Val`, and `Cmp`. These types are uninhabited by any value, but they index values in the GADT `F`, and they are associated with other, inhabited types by two type functions. We turn to these type functions next.

We want an interpreter `sprintf` for this domain-specific language, so that:

```
sprintf :: F f -> SPrintf f
```

where `SPrintf` is a type function that transforms the (type-level) format descriptor `f` to the type of `(sprintf f)`. For example, the following should all work:

```
sprintf f_ld          -- Result: "day"
sprintf f_lds         -- Result: "days"
sprintf f_dn 3        -- Result: "day 3"
sprintf f_nds 3       -- Result: "3 days"
```

It turns out that the most convenient approach is to use continuation-passing style, at both the type level and the value level. At the type level, we define `SPrintf` above using an auxiliary type function `TPrinter`. Because `TPrinter` has no accompanying value-level operations, a type class is not needed. Instead, GHC allows the type function to be defined directly, like this:[5]

```
type SPrintf f = TPrinter f String

type family TPrinter f x
type instance TPrinter L         x = x
type instance TPrinter (V val)   x = val -> x
type instance TPrinter (C f1 f2) x = TPrinter f1 (TPrinter f2 x)
```

So `SPrintf` is actually just a vanilla type synonym, which calls the type function `TPrinter` with second parameter `String`. Then `TPrinter` transforms the type as required. For example:

```
SPrintf (C L (V Int))  ~  TPrinter (C L (V Int)) String
                       ~  TPrinter L (TPrinter (V Int) String)
                       ~  TPrinter (V Int) String
                       ~  Int -> String
```

At the value level, we proceed thus:

[5] GHC requires the alarming flag `-XAllowUndecidableInstances` to accept the `(C f1 f2)` instance for `TPrinter`, because the *nested* recursive call to `TPrinter` does not "obviously terminate". Of course, every call to `TPrinter` does terminate, because the second argument (where the nested recursive call is made) is not scrutinised by any of the equations, but this is a non-local property that GHC does not check. The flag promises the compiler that `TPrinter` will terminate; the worst that can happen if the programmer makes an erroneous promise is that the type checker diverges.


```
sprintf :: F f -> SPrintf f
sprintf p = printer p id

printer :: F f -> (String -> a) -> TPrinter f a
printer (Lit str)    k = k str
printer (Val _ show) k = \x -> k (show x)
printer (Cmp f1 f2)  k = printer f1 (\s1 ->
                         printer f2 (\s2 ->
                         k (s1++s2)))
```

It is interesting to see how `printer` type-checks. Inside the `Lit` branch, for example, we know that `f` is `L`, and hence that the desired result type

`TPrinter f a` is `TPrinter L a`, or just `a`. Since `k str :: a`, the actual result type matches the desired one. Similar reasoning applies to the `Val` and `Cmp` branches.

4.2 Typed sscanf

We can use the same domain-specific language of format descriptors for parsing as well as printing. That is, we can write

```
sscanf :: F f -> SScanf f
```

where `SScanf` is a suitable type function. For example, reusing the format descriptors defined above, we may write:

```
sscanf f_ld  "days long"   -- Result: Just ((), "s long")
sscanf f_ld  "das long"    -- Result: Nothing
sscanf f_lds "days long"   -- Result: Just ((), " long")
sscanf f_dn  "day 4."      -- Result: Just (((),4), ".")
```

In general, `sscanf f s` returns `Nothing` if the parse fails, and `Just (v,s')` if it succeeds, where `s'` is the unmatched remainder of the input string, and `v` is a (left-nested) tuple containing the parsed values. The details are now fairly routine:

```
type SScanf f = String -> Maybe (TParser f (), String)

type family TParser f x
type instance TParser L        x = x
type instance TParser (V val)  x = (x,val)
```

```
type instance TParser (C f1 f2) x = TParser f2 (TParser f1 x)

sscanf :: F f -> SScanf f
sscanf fmt inp = parser fmt () inp

parser :: F f -> a -> String -> Maybe (TParser f a, String)
parser (Lit str)     v s = parseLit str v s
parser (Val reads _) v s = parseVal reads v s
parser (Cmp f1 f2)   v s = case parser f1 v s of
                             Nothing -> Nothing
                             Just (v1,s1) -> parser f2 v1 s1

parseLit :: String -> a -> String -> Maybe (a, String)
parseLit str v s = case prefix str s of
                     Nothing -> Nothing
                     Just s' -> Just (v, s')
```



```
parseVal :: Parser b -> a -> String -> Maybe ((a,b), String)
parseVal reads v s = case reads s of
                       [(v',s')] -> Just ((v,v'),s')
                       _         -> Nothing
```

4.3 Reflections

We conclude with a few reflections on the design.

- Our `Val` constructor makes it easy to add printers for new types. For example:

```
newtype Dollars = MkD Int

dollars :: F (V Dollars)
dollars = Val read_dol show_dol
  where
    read_dol ('$':s) = [ (MkD d, s) | (d,s) <- reads s ]
    read_dol _       = []
    show_dol (MkD d) = '$' : show d
```

- Our approach is precisely that of Hinze [20], except that we use type functions and GADTs (unavailable when Hinze wrote) to produce a much more elegant result.

- It is (just) possible to take the domain-specific-language approach *without* using type functions, albeit with less clarity and greater fragility [31].

- Defining `F` as a GADT makes it easy to define new interpreters beyond `sprintf` and `sscanf`, but hard to add new format-descriptor combinators. A dual approach [33], which makes it easy to add new descriptors but hard to define new interpreters, is to define `F` as a record of operations:

```
data F f = F {
  printer :: forall a. (String -> a) -> TPrinter f a,
  parser  :: forall a. a -> String
```

```
                                -> Maybe (TParser f a, String) }
```

Instead of being a GADT, `F` becomes a higher-rank data constructor – that is, its arguments are polymorphic functions. The type functions `TPrinter` and `TParser` are unchanged. The format-descriptor combinators are no longer data constructors but ordinary functions instead:

```
lit :: String -> F I
lit str = F { printer = \k -> k str,
              parser  = parseLit str }

int :: F (V Int)
int = F { printer = \k i -> k (show i),
          parser  = parseVal reads }
```

- If we consider only `sprintf` or only `sscanf`, then the type-level format descriptor is the result of *defunctionalizing* a type-level function, and `TPrinter` or `TParser` is the *apply function* [10, 39]. Considering

`sprintf` and `sscanf` together takes format descriptors out of the image of defunctionalization.

In general, type functions let us easily express a parser that operates on types (and produces corresponding values). In this way, we can overlay our own domain-specific, variable-arity syntax onto Haskell's type system.[6] For example, we can concisely express XML documents,[7] linear algebra,[8] and even keyword arguments.[9]

5 Fun with phantom types

Each type function above returns types that are actually used in the value-level computations. In other words, type functions are necessary to type-check the overloaded functions above. For example, it is thanks to the type function `Ref` that the value functions `newIORef` and `newSTRef` can be overloaded under the name `newRef`. In contrast, this section considers type functions that operate on so-called *phantom types*.

Phantom types enforce distinctions among values with the same run-time representation, such as numbers with different units [30] and strings for different XML elements. Functions on phantom types propagate these distinctions through a static approximation of the computation. Phantom types and functions on them thus let us reason more precisely about a program's behaviour before running it, essentially by defining additional type-checking rules that refine Haskell's built-in ones. The reader may find many applications of phantom types elsewhere [13, 14, 21]; our focus here is on the additional expressiveness offered by type families – to exclude more bad programs.

5.1 Pointer arithmetic and alignment

The refined distinctions afforded by phantom types are especially useful in embedded and systems programming, where a Haskell program (or code it generates) performs low-level operations such as direct memory access and interacts with hardware device drivers [11, 32]. It is easy to use phantom types to enforce access permissions (read versus write), but we take the example of pointer arithmetic and alignment to illustrate the power of type functions.

Many hardware operations require pointers that are properly aligned (that is, divisible) by a statically known small integer, even though every pointer, no matter how aligned, is represented by a machine word at run time. Our goal is to distinguish the types of differently aligned pointers and thus prevent the use of misaligned pointers.

Before we can track pointer alignment, we first need to define natural numbers at the type level. The type `Zero` represents 0, and if the type `n` represents n then the type `Succ n` represents $n + 1$.

[6] `http://okmij.org/ftp/Haskell/types.html#polyvar-fn`
[7] `http://okmij.org/ftp/Haskell/typecast.html#solving-read-show`
[8] `http://okmij.org/ftp/Haskell/typecast.html#is-function-type`
[9] `http://okmij.org/ftp/Haskell/keyword-arguments.lhs`


```
data Zero
data Succ n
```

For convenience, we also define synonyms for small type-level numbers.

```
type One   = Succ Zero
type Two   = Succ One
type Four  = Succ (Succ Two )
type Six   = Succ (Succ Four)
type Eight = Succ (Succ Six )
```

These type-level numbers belong to a class `Nat`, whose value member `toInt` lets us read off each number as an `Int`:

```
class Nat n where
  toInt :: n -> Int
instance Nat Zero where
  toInt _ = 0
instance (Nat n) => Nat (Succ n) where
  toInt _ = 1 + toInt (undefined :: n)
```

In this code, `toInt` uses a standard Haskell idiom called *proxy arguments*. As the underscores in its `instance`s show, `toInt` never examines its argument. Nevertheless, it must *take* an argument, as a proxy that specifies which instance to use. Here is how one might call `toInt`:

```
Prelude> toInt (undefined :: Two)
2
```

We use Haskell's built-in `undefined` value, and specify that it has type `Two`, thereby telling the compiler which instance of `Nat` to use. There is exactly such a call in the `(Succ n)` instance of `Nat`, only in that case the proxy argument is given the type `n`, a lexically scoped type variable.

As promised above, we represent a pointer or offset as a machine word at run time, but use a phantom type at compile time to track how aligned we know the pointer or offset to be.

```
newtype Pointer n = MkPointer Int
newtype Offset  n = MkOffset  Int
```

Thus a value of type `Pointer n` is an `n`-byte-aligned pointer; and a value of type `Offset n` is a multiple of `n`. For example, a `Pointer Four` is a 4-byte-aligned pointer. `Pointer n` is defined as a `newtype` and so the data constructor `MkPointer` has no run-time representation. In other words, the phantom-type alignment annotation imposes no run-time overhead.

To keep this alignment knowledge sound, the data constructors `MkPointer` and `MkOffset` above must not be exported for direct use by clients. Instead, clients must construct `Pointer` and `Offset` values using "smart constructors". One such constructor is `multiple`:

```
multiple :: forall n. (Nat n) => Int -> Offset n
multiple i = MkOffset (i * toInt (undefined :: n))
```

So `(multiple i)` is the `i`-th multiple of the alignment specified by the return type. For example, evaluating `multiple 3 :: Offset Four` yields `MkOffset 12`, the `3`rd multiple of a `Four`-byte alignment.

When a pointer is incremented by an offset, the resulting pointer is aligned by the greatest common divisor (GCD) of the alignments of the original pointer and the offset. To express this fact, we define a type function `GCD` to compute the GCD of two type-level numbers. Actually, `GCD` takes three arguments: `GCD d m n` computes the GCD of `d+m` and `d+n`. We will define `GCD` in a moment, but assuming we have it we can define `add`:

```
add :: Pointer m -> Offset n -> Pointer (GCD Zero m n)
add (MkPointer x) (MkOffset y) = MkPointer (x + y)
```

Thus, if `p` has the type `Pointer Eight` and `o` has the type `Offset Six`, then `add p o` has the type `Pointer Two`.

The type checker does not check that `x + y` is indeed aligned by the GCD. Like `multiple`, the function `add` is trusted code, and its type expresses claims that its programmer must guarantee. Once she does so, however, the clients of `add` have complete security. If `fetch32` is an operation that works on 4-aligned pointers only, then we can give it the type

```
(GCD Zero n Four ~ Four) => Pointer n -> IO ()
```

In words, `fetch32` works on any pointer whose alignment's GCD with 4 is 4. It is then a type error to apply `fetch32` to `add p o`, but it is acceptable to apply `fetch32` to `p`.

Because the type function `GCD` has no accompanying value-level operations, we can define it without a type class:

```
type family GCD d m n
type instance GCD d Zero Zero = d
```

```
type instance GCD d (Succ m) (Succ n) = GCD (Succ d) m n
type instance GCD Zero (Succ m) Zero = Succ m
type instance GCD (Succ d) (Succ m) Zero = GCD (Succ Zero) d m
type instance GCD Zero Zero (Succ n) = Succ n
type instance GCD (Succ d) Zero (Succ n) = GCD (Succ Zero) d n
```

5.2 Tracking state and control in a parameterised monad

Because actions in Haskell are values as well, phantom types can be used to enforce properties on actions and control flow as well as on values and data flow. In particular, we can express the preconditions and postconditions of monadic actions by generalising monads to *parameterised monads* [2]. A parameterised monad is a type constructor that takes three arguments, reminiscent of a Hoare triple: an initial state, a final state, and the type of values produced by the action. As shown in the following class definition (generalising the `Monad` class), a pure action does not change the state, and concatenating two actions identifies the final state of the first action with the initial state of the second action.

```
class PMonad m where
  unit :: a -> m p p a
  bind :: m p q a -> (a -> m q r b) -> m p r b
```

The precise meaning of states depends on the particular parameterised monad: they could describe files open, time spent, or the shape of a managed heap [32]. In this example, we use a parameterised monad to track the locks held among a given (finite) set.

A lock can be acquired only if it is not currently held, and released only if it is currently held. Furthermore, no lock is held at the beginning of the program, and no lock should be held at the end. We encode a set of locks and whether each is held by a type-level list of booleans. The spine of the list is made of `Cons` cells and `Nil`; each element of the list is either `Locked` or `Unlocked`. For example, suppose we are tracking three locks. If only the first and last are held, then the state is the type `Cons Locked (Cons Unlocked (Cons Locked Nil))`.

```
data Nil
data Cons l s

data Locked
data Unlocked
```

The run-time representation of our parameterised monad is simply that of Haskell's `IO` monad, so it is easy to implement a `PMonad` instance.

```
newtype LockM p q a = LockM { unLockM :: IO a }

instance PMonad LockM where
  unit x   = LockM (return x)
  bind m k = LockM (unLockM m >>= unLockM . k)
```

It is also easy to lift an `IO` action that does not affect locks to become a `LockM` action whose initial and final states are the same and arbitrary.

```
lput :: String -> LockM p p ()
lput = LockM . putStrLn
```

To manipulate boolean lists at the type level, we define type functions `Get` and `Set`. Given a type-level natural number `n` and a list `p`, the type `Get n p` is the `n`-th element of that list, and the type `Set n e p` is the result of replacing the `n`-th element of `p` by `e`. The first element of a list is indexed by `Zero`. It is a type error if the element does not exist because the list is too short.

```
type family Get n p
type instance Get Zero (Cons e p) = e
type instance Get (Succ n) (Cons e p) = Get n p

type family Set n e' p
type instance Set Zero e' (Cons e p) = Cons e' p
type instance Set (Succ n) e' (Cons e p) = Cons e (Set n e' p)
```

We represent a lock as a mutex handle (here caricatured by an `Int`), with a phantom type `n` attached to identify the lock at compile time. The phantom type `n` is an index into a type-level list.

```
newtype Lock n = Lock Int deriving Show
```

```
mkLock :: forall n. Nat n => Lock n
mkLock = Lock (toInt (undefined::n))
```

The data constructor introduced by the `newtype` declaration has no run-time representation and so this wrapping imposes no run-time overhead. We make one lock, `lock1`, for the sake of further examples.

```
lock1 = mkLock :: Lock One
```

We can now define actions to acquire and release locks. The types of the actions reflect their constraints on the state.

```
acquire :: (Get n p ~ Unlocked) =>
           Lock n -> LockM p (Set n Locked p) ()
acquire l = LockM (putStrLn ("acquire " ++ show l))

release :: (Get n p ~ Locked) =>
           Lock n -> LockM p (Set n Unlocked p) ()
release l = LockM (putStrLn ("release " ++ show l))
```

In the type of `acquire`, the constraint `Get n p ~ Unlocked` is the *precondition* on the state before acquiring the lock: the lock to be acquired must not be already held. The final state of the `LockM` action returned by `acquire` specifies the *postcondition*: the lock just acquired is `Locked`. For the `release` action, the pre- and postconditions are the converse. To keep the example simple, we do not manipulate any real locks; rather, we print our intentions.

At the top level, a `LockM` action is executed by applying the function `run` to turn it into an `IO` action. The type of `run` below requires that the action begin and end with no lock held among three available.

```
type ThreeLocks = Cons Unlocked (Cons Unlocked (Cons Unlocked Nil))
run :: LockM ThreeLocks ThreeLocks a -> IO a
run = unLockM
```

For example, given any action `a`, the action `with1 a` defined below acquires lock 1, performs `a`, then releases lock 1 and returns the result of `a`.

```
with1 a = acquire lock1 `bind` \_ ->
          a `bind` \x ->
          release lock1 `bind` \_ ->
          unit x
```

Therefore, we can execute `run (with1 (lput "hello"))` by itself.

```
> run (with1 (lput "hello"))
acquire Lock 1
hello
release Lock 1
```

Multiple locks can be held at the same time and need not be released in the opposite order as they were acquired. However, the type system prevents us from nesting `with1` inside `with1`, because such an action would try to acquire lock 1 twice. Indeed, the expression `run (with1 (with1 (lput "hello")))` does not type-check. We also cannot acquire a lock without releasing it subsequently. For example, the expression `run (acquire lock1)` is rejected.

We can also introduce actions that do not change the state of locks yet require that a certain lock be held:

```
critical1 :: (Get One p ~ Locked) => LockM p p ()
critical1 = LockM (putStrLn "Critical section 1")
```

An attempt to run such an action without holding the required lock, as in `run critical1`, is rejected by the type checker. On the other hand, the program `run (with1 critical1)` type checks and can be successfully executed. Likewise, we can define potentially blocking actions, to be executed only when a lock is not held; the type checker will then prevent such actions within a critical section protected by the lock.

5.3 Keeping the kinds straight

It will not have escaped the reader's notice that we are doing *untyped* functional programming at the type level. For example, the kind of `GCD` is

```
GCD :: * -> * -> * -> *
```

so the compiler would accept the nonsensical type `(GCD Int Zero Bool)`. The same problem occurs with `Pointer n` and other types defined in this section. We can alleviate the problem using the `Nat n` constraint. For example, we could define `Pointer n` as

```
newtype Nat n => Pointer n = MkPointer Int
```

so that, for example, `Pointer Bool` becomes invalid and will raise a compile-time error. The constraint `Nat n` is a *kind predicate*, specifying the set of types that constitute natural numbers – just as the type `Int` specifies a set of values.

We wish for the convenience and discipline of algebraic data kinds when writing type-level functions, just as we are accustomed to algebraic data types in conventional, term-level programs. We could find a way to 'lift' the ordinary data type declaration

```
data N = Zero | Succ N
```

to the kind level. Alternatively, we may want to declare algebraic data kinds like this:

```
data kind N = Zero | Succ N
```

Here `N` is a kind constant and `Zero` and `Succ` are type constructors. Now `GCD` could have the kind

```
GCD :: N -> N -> N -> N
```

Similarly, `Pointer` and `Offset` should both have kind `N -> *`. Much the same applies in the discussion of state and control, where we would rather write:

```
data kind ListLS = Nil | Cons LockState ListLS
```

```
data kind LockState = Locked | Unlocked
```

then give a decent kind to `Get`:

```
Get :: N -> ListS -> LockState
```


Furthermore, unlike the earlier examples in which it was crucial that our type functions were open (Section 2.5), type functions such as `GCD` and `Get` are *closed*, in that all their equations are given in one place.

These are shortcomings of GHC's current implementation, but there is no technical difficulty with algebraic data kinds, and indeed they are fully supported by the Omega language [43].

5.4 Type-preserving compilers

A popular, if incestuous, application of Haskell is for writing compilers. If the object language is statically typed, then one can index a GADT by a phantom type to ensure that only well-typed object programs can be represented in the compiler [37]:

```
data Exp a where
  Enum :: Int -> Exp Int
  Eadd :: Exp Int -> Exp Int -> Exp Int
  Eapp :: Exp (a->b) -> Exp a -> Exp b
  ...
```

Now an optimiser and an evaluator might have types

```
optimise :: Exp a -> Exp a
evaluate :: Exp a -> a
```

which compactly express the facts that (a) the optimiser need only deal with well-typed object terms, (b) optimising a term does not change its type, and (c) evaluating a term yields a value of the correct type.

But what about transforming programs into continuation-passing style? In that case, the type of the result term is a *function of* the type of the argument term:

```
cpsConvert :: Exp a -> Exp (CpsT a)
```

Here `CpsT` maps a type `a` to its CPS-converted version [34]. Guillemette and Monnier express `CpsT` as a type-level function [18], whereas Carette et al. show how to do without type-level functions [4].

6 Related work and reflections

The goal of type families is to build on the success of static type systems, by extending their power and expressiveness without losing their brevity and comprehensibility to programmers. (Of course, there is an implicit tension between these goals, and the reader will have to judge how successful we have been.) There are other designs with similar goals:

- Functional dependencies took the Haskell community by storm when Mark Jones introduced them [29], because they met a real need.

Many, perhaps all, of the examples in this tutorial can also be programmed using functional dependencies, but the programming style at the type level feels like logic programming rather than functional programming. The reader may find a programmer's-eye comparison of the two approaches in [6]. Jones showed recently how the stylistic question can be at least partly addressed by a notational device [28]

but, more fundamentally, the interaction of functional dependencies with other type-level features such as existentials and GADTs is not well understood and possibly problematic. In fact, one may see type families as a way to understand functional dependencies in these more general settings.

- Omega [43] is a prototype programing language that specifically aims to provide the programmer with type-level computation. It goes quite a bit further than GHC's type families (for example, Omega has an infinite tower of kinds and supports closed type functions), but lacks type classes and much of the other Haskell paraphernalia. Omega comes with a number of excellent papers giving many a motivating example [44–46].

These designs, along with GHC's type families, can be thought of as helping programmers prove more interesting theorems that characterise their programs. Meanwhile, the theorem-proving and type-theory community has been drawing from its long history of type-level computation to help mathematicians write more interesting programs that witness their theorems [3].

The motivation for type-level computations comes from the Curry-Howard correspondence [17, 23] that underlies Martin-Löf's intuitionistic type theory: propositions are types, and proofs are terms. The more expressive a type system, the more propositions we can state and prove in it, such as properties involving numbers and arithmetic. Hence expressive languages such as those of NuPRL, Coq, Epigram, and Agda permit types involving numbers and arithmetic. For example, the following type in Agda states that addition is commutative:

```
(n m : Nat) -> n + m == m + n
```

To prove this proposition is to write a term of this type, and to check the proof is the job of the type checker. To do its job, the type checker may need to simplify a type like `(Zero + m)` to `m`, so type checking involves type-level computations. Because a proof checker should always terminate, it is natural to insist that type-level computations also always terminate.

Since proof assistants based on type theory implement a (richly typed) λ-calculus, they can be used to program – that is, to write terms that compute interesting values, not just inhabit interesting types. To this end, an expressive type system lets us state and prove more interesting properties about programs – of the sort we have shown in this paper. Tools such as Coq, Epigram, and Agda thus cater increasingly to the use of theorem proving for practical programming. This convergence of theory and practice renews our commitment to Tony Hoare's ideal of simple, reliable software.

Acknowledgements

We would like to thank people who responded to our invitation to suggest interesting examples of programming with type families, or commented

on a draft of the paper: Lennart Augustsson, Neil Brown, Toby Hutton, Ryan Ingram, Chris Kuklewicz, Dave Menendez, Benjamin Moseley, Hugh Pacheco, Conrad Parker, Bernie Pope, Tom Schrijvers, Josef Svenningsson, Paulo Tanimoto, Magnus Therning, Ashley Yakeley, and Brent Yorgey.

References

[1] Asai, Kenichi. 2008. On typing delimited continuations: three new solutions to the printf problem. Tech. Rep. OCHA-IS 08-2. `http://pllab.is.ocha.ac.jp/~asai/papers/tr07-1.ps.gz`.

[2] Atkey, Robert. 2009. Parameterised notions of computation. *Journal of Functional Programming* 19:355–376.

[3] Bove, Anna, and Peter Dybjer. 2009. Dependent types at work. In *International summer school on language engineering and rigorous software development*. Lecture Notes in Computer Science 5520.

[4] Carette, Jacques, Oleg Kiselyov, and Chung-chieh Shan. 2008. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*. In press.

[5] Chakravarty, Manuel. 2008. Type families. `http://haskell.org/haskellwiki/GHC/Indexed_types`.

[6] Chakravarty, Manuel M. T., Gabriele Keller, and Simon L. Peyton Jones. 2005. Associated type synonyms. In *ICFP '05: Proc. ACM international conference on functional programming*, 241–253. New York: ACM Press.

[7] Chakravarty, Manuel M. T., Gabriele Keller, Simon L. Peyton Jones, and Simon Marlow. 2005. Associated types with class. In *POPL '05: Conference record of the annual ACM symposium on principles of programming languages*, ed. Jens Palsberg and Martín Abadi, 1–13. New York: ACM Press.

[8] Cunha, Alcino, Jorge Sousa Pinto, and José Proença. 2006. A framework for point-free program transformation. In *Revised selected papers from IFL 2005: Implementation and application of functional*

languages, ed. Andrew Butterfield, Clemens Grelck, and Frank Huch, 1–18. Lecture Notes in Computer Science 4015, Berlin: Springer.

[9] Danvy, Olivier. 1998. Functional unparsing. *Journal of Functional Programming* 8(6):621–625.

[10] Danvy, Olivier, and Lasse R. Nielsen. 2001. Defunctionalization at work. In *Proceedings of the 3rd international conference on principles and practice of declarative programming*, 162–174. New York: ACM Press.

[11] Diatchki, Iavor S., and Mark P. Jones. 2006. Strongly typed memory areas: Programming systems-level data structures in a functional language. In *Proceedings of the 2006 Haskell Workshop*. New York: ACM Press.

[12] Elliott, Conal. 2008. Elegant memoization with functional memo tries. `http://conal.net/blog/posts/elegant-memoization-with-functional-memo-tries`.

[13] Fluet, Matthew, and Riccardo Pucella. 2005. Practical datatype specializations with phantom types and recursion schemes. In *Proceedings of the 2005 workshop on ML*. Electronic Notes in Theoretical Computer Science.

[14] ———. 2006. Phantom types and subtyping. *Journal of Functional Programming* 16(6):751–791.

[15] Garcia, Ronald, Jaakko Jarvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. 2007. An extended comparative study of language support for generic programming. *Journal of Functional Programming* 17(2):145–205.

[16] Gill, Andrew, ed. 2008. *Proceedings of the 1st ACM SIGPLAN symposium on Haskell*. New York: ACM Press.

[17] Girard, Jean-Yves, Paul Taylor, and Yves Lafont. 1989. *Proofs and types*. Cambridge: Cambridge University Press.

[18] Guillemette, Louis-Julien, and Stefan Monnier. 2008. A type-preserving compiler in Haskell. In [25], 75–90.

[19] Hinze, Ralf. 2000. Generalizing generalized tries. *Journal of Functional Programming* 10(4):327–351.

[20] ———. 2003. Formatting: A class act. *Journal of Functional Programming* 13(5):935–944.

[21] ———. 2003. Fun with phantom types. In *The fun of programming*, ed. Jeremy Gibbons and Oege de Moor, 245–262. Palgrave.

[22] Hinze, Ralf, Johan Jeuring, and Andres Löh. 2002. Type-indexed data types. In *Proceedings of the Sixth International Conference on Mathematics of Program Construction (MPC 2002)*, 148–174. Lecture Notes in Computer Science 2386, Springer Verlag.

[23] Howard, William A. 1980. The formulae-as-types notion of construction. In *To H. B. Curry: Essays on combinatory logic, lambda calculus and formalism*, ed. Jonathan P. Seldin and J. Roger Hindley, 479–490. San Diego, CA: Academic Press.

[24] Hutton, Toby. 2008. Fun with type functions. `http://www.haskell.org/pipermail/haskell-cafe/2008-November/051105.html`.

[25] ICFP08. 2008. *ICFP '08: Proc. ACM international conference on functional programming*. New York: ACM Press.

[26] Imai, Keigo, Shoji Yuen, and Kiyoshi Agusa. 2009. A full implementation of session types in haskell. In *PPL2009: 11th programming and programming languages workshop*. `http://www.agusa.i.is.nagoya-u.ac.jp/person/sydney/fullsession-ppl2009/20090224/imai-ppl2009-submitted1.pdf`.

[27] Ingram, Ryan. 2008. Fun with type functions. `http://www.haskell.org/pipermail/haskell-cafe/2008-November/051108.html`.

[28] Jones, Mark. 2008. Languages and program design for functional dependencies. In [16], 87–98.

[29] Jones, Mark P. 2000. Type classes with functional dependencies. In *Programming Languages and Systems: Proceedings of ESOP 2000, 9th European Symposium on Programming*, ed. Gert Smolka, 230–244. Lecture Notes in Computer Science 1782, Berlin: Springer.

[30] Kennedy, Andrew. 1995. Programming languages and dimensions. Ph.D. thesis, University of Cambridge.

[31] Kiselyov, Oleg. 2008. Formatted IO as an embedded DSL: the initial view. `http://okmij.org/ftp/typed-formatting/#DSL-In`.

[32] Kiselyov, Oleg, and Chung-chieh Shan. 2007. Lightweight static resources: Sexy types for embedded and systems programming. In *Draft Proceedings of TFP 2007: 6th Symposium on Trends in Functional Programming*, ed. Marco T. Morazán and Henrik Nilsson. Tech. Rep. TR-SHU-CS-2007-04-1, Department of Mathematics and Computer Science, Seton Hall University.

[33] Krishnamurthi, Shriram, Matthias Felleisen, and Daniel P. Friedman. 1998. Synthesizing object-oriented and functional design to promote re-use. In *Proceedings of ECCOP'98: 12th European conference on object-oriented programming*, ed. Eric Jul, 91–113. Lecture Notes in

Computer Science 1445, Berlin: Springer.

[34] Meyer, Albert R., and Mitchell Wand. 1985. Continuation semantics in typed lambda-calculi (summary). In *Logics of programs*, ed. Rohit Parikh, 219–224. Lecture Notes in Computer Science 193, Berlin: Springer.

[35] Michie, Donald. 1968. "Memo" functions and machine learning. *Nature* 218:19–22.

[36] Neubauer, Matthias, and Peter Thiemann. 2004. An implementation of session types. In *Practical Aspects of Declarative Languages: 6th International Symposium, PADL 2004*, ed. Bharat Jayaraman, 56–70. Lecture Notes in Computer Science 3057, Berlin: Springer.

[37] Peyton Jones, Simon L., Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Alan Washburn. 2006. Simple unification-based type inference for GADTs. In *ICFP '06: Proc. ACM international conference on functional programming*, 50–61. New York: ACM Press.

[38] Pucella, Riccardo, and Jesse Tov. 2008. Haskell session types with (almost) no class. In [16], 25–36.

[39] Reynolds, John C. 1972. Definitional interpreters for higher-order

programming languages. In *Proceedings of the ACM National Conference*, vol. 2, 717–740. New York: ACM Press. Reprinted as [40].

[40] ———. 1998. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation* 11(4):363–397.

[41] Sackman, Matthew. 2008. A tutorial for session types. `http://www.wellquite.org/sessions/tutorial_1.html`.

[42] Schrijvers, Tom, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. 2008. Type checking with open type functions. In [25], 51–62.

[43] Sheard, Tim. 2004. Languages of the future. *Onward Track, OOPSLA'04. Reprinted in: ACM SIGPLAN Notices, Dec. 2004.* 39:116–119. OOPSLA Companion Volume.

[44] ———. 2006. Generic programming programming in Omega. In *Datatype-generic programming*, ed. Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, vol. 4719 of *Lecture Notes in Computer Science*, 258–284. Springer.

[45] Sheard, Tim, and Nathan Linger. 2007. Programming in Omega. In *2nd Central European Functional Programming School*, ed. Zoltán Horváth, Rinus Plasmeijer, Anna Soós, and Viktória Zsók, vol. 5161

of *Lecture Notes in Computer Science*, 158–227. Springer.

[46] Sheard, Tim, and Emir Pasalic. 2004. Meta-programming with built-in type equality. In *Proceedings of the fourth international workshop on logical frameworks and meta-languages (LFM'04)*.

[47] Siek, Jeremy, Lie-Quan Lee, and Andrew Lumsdaine. 2001. *The Boost Graph Library User Guide and Reference Manual*. Addison-Wesley.

Appendices

These appendices will not appear in the published paper, only in the online version.

A The Rules

Here we summarise some rules governing type families. The reader may find more details elsewhere [5–7, 42].

The *indices* of a type family are the arguments that appear to the left of the "::" in its kind signature.

1. Like ordinary Haskell **type** synonyms, a **type** family must always be saturated; that is, it must be applied to all its type indices. For example:

   ```
   data D m = MkD (m Int)      -- So D :: (*->*) -> *

   type family T a :: *        -- So T :: * -> *
   f1 :: D T                   -- ILLEGAL (unsaturated)

   type family S a :: * -> *   -- So S :: * -> * -> *
   f2 :: D (S a)               -- OK (saturated)

   type family R a b :: *      -- So R :: * -> * -> *
   f3 :: D (R a)               -- ILLEGAL (unsaturated)
   ```

   This constraint does not apply to `data` families.

2. In a `type instance` or `data instance` declaration, any arguments that are not type indices must be type variables. For example:

   ```
   type family T a :: * -> *
   type instance T Int b    = Int   -- OK
   type instance T Int Bool = Int   -- Not allowed
   type instance T a   Bool = Int   -- Not allowed
   ```

3. In an associated `type` or `data` declaration (i.e. one appearing nested in a `class` declaration), the type indices must be a permutation of one or more of the class variables. For example:

   ```
   class C a b where
      type T1 a   :: *        -- OK
   ```

```
type T2 b   :: *        -- OK
type T3 b a :: *        -- OK
type T4 a c :: *        -- Not OK; mentions 'c'
type T5 a   :: * -> *   -- OK
```

4. There is no difference between a type family declared as an associated type of a class declaration, and a type family declared at top level. For example, the following are equivalent:

```
class C1 a b where          |    type family T2 a :: *
  type T1 a :: *            |    class C2 a b where
  op :: a -> b -> Int       |      op :: a -> b -> Int
instance C1 Int Int where   |    type instance T2 Int = Bool
  type T1 Int = Bool        |    instance C2 Int Int where
  op = ...                  |      op = ...
```

B Pitfalls

Type functions are powerful, but they can give rise to unexpected errors. In this appendix we review some of the more common cases.

B.1 Ambiguity

One pitfall of type functions commonly mentioned on Haskell mailing lists is a false expectation that they are injective. As we discussed in Section 2.4, type functions are, in general, not injective: if `F` is a type family, then the fact `F t1` is the same as `F t2` does not imply that `t1` and `t2` are the same (that fact is easy to see for the type function mapping any type to `Int`). Therefore, the type checker cannot use the equality of `F t1` and `F t2` to equate `t1` and `t2`. The pitfall of the false expectation of injectivity of type functions can be quite subtle. Consider the following example (abstracted from a recent message on the Haskell-Cafe mailing list):

```
class C a where
  type F a :: *
  inj :: a -> F a
  prj :: F a -> a

-- bar :: (C a) => F a -> F a
bar x = inj (prj x)
```

That code type-checks; the inferred type signature is given in the comments. The signature agrees with our expectation. If we uncomment the signature, the type-checking fails:

```
foo.hs:8:17:
    Couldn't match expected type ‘F a’ against inferred type ‘F a1’
    In the first argument of ‘prj’, namely ‘x’
```

It seems GHC does not like the signature it itself inferred! In fact, the bug here is that GHC should not have accepted the signature-less `bar` in the first place, because `bar` embodies an unresolvable ambiguity. To see the problem clearly, let us assume the following instances of the class `C`:

```
instance C Int where
    type F Int = Int
    inj = id
    prj = id

instance C Char where
    type F Char = Int
    inj _ = 0
    prj _ = 'a'
```

Given the application `bar (1::Int)`, which instance of `prj` should the compiler choose: `prj:: Int -> Int` or `prj:: Int -> Char`? The choice determines the result of `bar 1`: 1 or 0, respectively. The application `bar (1::Int)` provides no information to help make this choice; in fact, *no* context of `bar` usage can resolve the ambiguity. The function `bar` is an instance of the infamous read-show problem, the composition `show . read`, which is just as ambiguous.

B.2 Lack of inversion

Even if a type function (defined as a type family rather than a data family) turns out to be injective, GHC will not notice that fact; in particular, GHC

will not try to invert such a type function. For example, we may easily define addition of type-level naturals (§5.1) as a type family

```
type family Plus m n
type instance Plus Zero n = n
type instance Plus (Succ m) n = Succ (Plus m n)

plus :: m -> n -> Plus m n
plus = undefined

tplus = plus (undefined::Two) (undefined::Three)
```

The expression `tplus` has the monomorphic inferred type `Plus Two Three` (with no constraints attached), and `toInt tplus` evaluates to `5`. One may expect that a related `tplus'`

```
tplus' x = if True then plus x (undefined::One) else tplus
```

will have a monomorphic type, too. However, GHC infers a polymorphic type with a type equality constraint:

```
tplus' :: (Succ (Succ (Succ Two)) ~ Plus m One) => m -> Plus m One
```

There is only a single type `m` (viz. `Four`) that satisfies the constraint; one might hope that GHC would figure it out and resolve the constraint. One should keep in mind that GHC is not a general-purpose solver for arithmetic and other constraints. The type families like `GCD` and `Plus` along with the type equality let us write types with arbitrary arithmetic constraints over unbounded domain of type-level natural numbers. Solving these constraints is an undecidable problem.

C Sprintf revisited

In this appendix we explore yet another variant on `sprintf`, this one including *higher order* type-level functions. Recall that `sprintf` should take as an argument a format descriptor and zero or more additional arguments. The number and the type of the additional arguments – the values to format – depend on the type of the format descriptor. The function `sprintf` should return the formatted string. A format descriptor is an expression built by connecting primitive descriptors such as `lit "str"` and `int` with a descriptor composition operator `(^)`. For example,

```
sprintf (lit "day")                          -- Result: "day",
sprintf (lit "day" ^ lit "s")                -- Result: "days",
sprintf (lit "day " ^ int) 3                 -- Result: "day 3",
sprintf (int ^ lit " day" ^ lit "s") 3       -- Result: "3 days"
```

The specification immediately suggests the following naive implementation. Since the format descriptor `lit "str"` denotes outputting (as the result of `sprintf`) of the string `str`, `lit "str"` may just as well be `str` itself. Thus `lit "str"` has the type `String`. The function `sprintf` is the identity then. The descriptor `int` denotes receiving an integer and outputting it as a string, hence `int` could be implemented as a function `show` of the type `Int->String`. The composition of the format descriptors should therefore concatenate the outputs of the descriptors. That is easy to do if the two descriptors are `lit "str1"` and `lit "str2"`, in which case we just concatenate `str1` and `str2`. When we compose `int` and `lit "str1"`, we would like the composite format descriptor to be `\x -> show x ++ "str"`. Thus, the left-associative composition of two descriptors is type-directed:

```
fmt1 ^ fmt2 = fmt1 ++ fmt2
  when fmt1 :: String and fmt2 :: String
fmt1 ^ fmt2 = \x -> fmt1 x ++ fmt2
  when fmt1 :: Int -> String and fmt2 :: String
fmt1 ^ fmt2 = \x -> fmt1 ++ fmt2 x
  when fmt1 :: String and fmt2 :: Int -> String
...
```

We have to analyse and induct on the types of both arguments of `(^)`.

We can change the representation of descriptors so that we need case analysis on the type of only one argument of `(^)`. In the naive implementation, format descriptors have the general type `t1 -> t2 -> ... -> String`. The composition of the two descriptors have to 'dive' under the layers of `t1 -> t2 -> ...` in order to concatenate the underlying `String`s – for *both* descriptors. Let us change the implementation: let `lit "str"` be a function that takes the current output as the string and appends to it `str`:

```
lit :: String -> (String -> String)
lit str = \s -> s ++ str
```

Likewise, `int` should receive the output so far, obtain an integer and return the new output, with the formatted integer appended to the current output:

```
int :: String -> Int -> String
int = \s -> \x -> s ++ show x
```

Thus the formatters have the general type `String -> t1 -> t2 -> ... -> String`. With this implementation of the formatter, the composition of formatters can be informally defined as

```
fmt1 ^ fmt2 = \s -> fmt2 (fmt1 s)
  when fmt1 :: String -> String and fmt2 :: String -> t
fmt1 ^ fmt2 = \s -> \x -> fmt2 (fmt1 s x)
  when fmt1 :: String -> Int -> String and fmt2 :: String -> t
...
```

The formatter composition operation `(^)` needs case analysis on the type of only one argument, which is straightforward with the help of an ordinary, one-parameter type class. Here is the first attempt:

```
class FCompose a where
    (^) :: (String -> a) -> (String -> b) -> (String -> ???)
```

What is the return type of `(^)` should be however? It is obvious that `???` must depend on both `a` and `b`. The informal definition shows that if `a` is `String`, `???` is just `b`. If `a` is `Int->String` however, then `???` is `Int -> b`. In general, if `a` is `t1 -> t2 -> ... String`, then `???` must be `t1 -> t2 -> ... b`. We can try to use associated type synonyms to express such a result type:


```
class FCompose a where
    type Result a :: * -> *
    (^) :: (String -> a) -> (String -> b) -> (String -> Result a b)

instance FCompose String where
    type Result String b = b
    (^) f1 f2 = ...

instance FCompose c => FCompose (a -> c) where
    type Result (a -> c) b = a -> Result c b
    (^) f1 f2 = ...
```

Alas, for technical reasons this attempt doesn't work: `Result a` is defined as having one type parameter and yielding an existing type (constructor or a function) of the kind `* -> *`. After all, `Result` is the type *synonym*. Therefore, the definition of `Result` associated with the instance `FCompose String` is invalid as `Result String` is not defined to be a synonym of an existing type constructor or a function of the kind `* -> *`. To get around that, we resort to type families, which are free from such restrictions. Here is the final, working implementation:

```
data I
class FCompose a where
```

```
  type Result a
  (^) :: (String -> a) -> (String -> b) ->
         (String -> TApply (Result a) b)

instance FCompose String where
  type Result String = I
  (^) f1 f2 = \s -> f2 (f1 s)

instance FCompose c => FCompose (a -> c) where
  type Result (a -> c) = a->Result c
  (^) f1 f2 = \s -> \x -> ((\s -> f1 s x) ^ f2) s
```

The associated type synonym `Result a` 'computes' a type function (more precisely, a functor) mapping a type `b` to a type containing `b`. To be precise, `Result a` is a type that *represents* a functor. The language of representations is trivial: the type `I` represents the identity functor, and the type `T1 -> I` represents the functor that takes a type `b` to a type `T1 -> b`. In other words, `T1 -> F` represents the *functional composition* of the functors `(T1 ->)` and `F`. The type family `TApply F x` *interprets* the mini-language of functor representations and performs the application of the corresponding functor to a type `x`:

```
type family TApply functor x
type instance TApply I x = x
type instance TApply (a -> c) x = a -> TApply c x
```

Essentially, `TApply` is a *higher-order* type function.

The function `sprintf` is then a simple wrapper over the format descriptor:

```
sprintf:: (String -> t) -> t
sprintf fmt = fmt ""
```