

Beginning Faking It

Uktaad B'mal

October 5, 2004

One of the topics that appeared over and over at the Haskell workshop was type-level programming. It seems that although Haskell lacks dependent types, there is still a lot of expressiveness in the type system. One person pointed out that Haskell is really a functional-logic language, functional at the value level, and a logic language at the type level.

Although this post doesn't go anywhere near the stuff some of the folks at the workshop were doing, it does intend to give a bit of an introduction to calculating at the type level. I have some references if anyone is interested. For example, Conor McBride describes this material in more detail [2].

1 Term-level Naturals

We can begin with a term-level definition of natural numbers, then later we look at type-level calculations.

The *Nats* are an inductively defined type, as you would expect.

```
data Nat = Succ Nat | Zero deriving Show
```

We write a couple of helper functions for moving between the *Nat* representation and Integers.

```
toNat 0 = Zero
```

```
toNat (n + 1) = Succ (toNat n)
```

```
fromNat Zero = 0
```

```
fromNat (Succ n) = 1 + (fromNat n)
```

Now we define addition inductively.

```
add Zero x = x
```

```
add x Zero = x
```

```
add (Succ x) y = Succ (add x y)
```

2 Type-level Naturals

We can now try to move this to the type level. First, we define two different *types*, *Z* and *S*. Recall that at the term level, we have two different terms, *Zero* and *Succ*, which build inhabitants of the *type* *Nat*. Now that we're moving to the type level, we have two different types, *Z* and *S*, which build inhabitants of the *kind* (the "type of a type") ***. Notice that we lose some restrictions that we had in at the term level, in that the type system prevented expressions like *Succ "Nada"*. However, at the type level, we don't partition the kind space, so type expressions like *S String* are permissible, in addition to expected valid type expressions like *S Z*. There was a mention by SPJ on the Haskell list about being able to enforce the "kind-space" partition in future GHC revisions, but I don't know what's come of it. Tim Sheard is working on a language, called Ω mega, which provides this capability.

```
data Z
```

```
data S a
```

2.1 Adding Types

Having defined types for naturals, the class system can be used to define an addition function for these types. This requires *functional dependencies*, an extension to Haskell98. Functional dependencies were introduced to help the type system resolve overloading for multi-parameter type classes [1]. In effect, functional dependencies allow us to describe partial functions between types. In the *Add* class below, the expression $a\ b \rightarrow c$ means “types a and b *uniquely* determine the type c ”.

The *Add* class is used to defined type-level addition. The class signature, *Add a b c* can be read as “ a plus b equals c ”. We define a single *typeAdd* method, but as we’ll see later, we don’t actually use this function to calculate values; all of the calculation is performed by the typechecker.

```
class Add a b c | a b → c where
  typeAdd :: a → b → c
  typeAdd _ _ = error "typeAdd"
```

The *Add* class involves two instance declarations. Calculating at the type level, using type classes, takes on a definite logic flavor. Compare the below instance declarations to an equivalent definition of *Add* in Prolog.

```
Add(zero, X, X).
Add(s X, Y, s Z) :- Add X Y Z
```

The definitions are nearly identical (if mirror images of each other). We don’t need to provide any definitions for *typeAdd*, because we’ll never actually evaluate it.

```
instance Add Z x x
instance Add x y z ⇒ Add (S x) y (S z)
```

2.2 Creating Values

One of the problems with the above definitions of the S and Z types is that they are completely abstract, so we don’t have any constructors for creating values of those types. Luckily, though, all types in Haskell are *lifted*: \perp inhabits every type. Additionally, \perp has the type $\forall a.a$, so we can give it any type we want with an explicit type annotation.

We define two values, *zero* and *one*, which give base values for type computations. Using these two values, and the addition definitions, we can create any natural.

```
zero :: Z
zero = ⊥
one  :: S Z
one  = ⊥
```

To calculate using types, we invoke the typechecker from the command line. The type of the resulting expression is the same (except it’s at the type level) as the equivalent term-level expression.

```
TypeAdd> :type typeAdd one one
TypeAdd> typeAdd oneone :: S (S Z)
```

2.3 Multiplication

We can define multiplication in much the same way as we defined addition. First, we give a term-level definition of the operation, then write a type-level version with a logic programming flavor.

The *mult* definition given calculates using an iterative application of the previously defined *add*. The second case is extraneous, but it exploits the identity of one in multiplication for efficiency. Additionally, the recursive case is written slightly more complicated, to make the transition to the type-level definition easier.

```
mult Zero x = Zero
mult (Succ Zero) x = x
mult (Succ x) y = let z = (mult x y)
```

```

    prod = add y z
  in prod

```

Below is the typeclass definition. We read this as "a times b equals c". We define the functional dependency between a and b and the product, c .

```

class Mult a b c | a b → c where
  typeMult :: a → b → c
  typeMult _ _ = error "Undefined"

instance Mult Z b Z
instance Mult (S Z) b b
instance (Mult x y z, Add y z prod) ⇒ Mult (S x) y prod

```

Now we can evaluate $3 * 2$ to get 6.

```

TyCalc>:type typeMult (undefined :: (S (S (S Z)))) (undefined :: S (S Z))
TyCalc>typeMult (undefined :: (S (S (S Z)))) (undefined :: S (S Z)) ::
  S (S (S (S (S (S Z))))))

```

Exercise 1 Define an exponentiation operator. First define it at the term level, then convert that to a type-level calculation.

2.4 Converting from types to Ints

One difficulty of with doing calculating with types is moving between the type and the term level. Earlier we defined *zero* and *one* values for creating the desired types. To go the other way, we can define a *reflection* function, which converts a give type into it's proper term representation. Once again, we don't actually use the *values* of the parameters of the functions; we just want to have access to their types. Note, in the second instance declaration, we make use of the type variable x in scope as an explicit type annotation.

```

class Reflect s where
  intValue :: s → Int

instance Reflect Z where
  intValue _ = 0

instance Reflect x ⇒ Reflect (S x) where
  intValue _ = intValue (⊥ :: x) + 1

```

Using the above definitions, we can reify the simple values, as well as the reified result of a type calculation.

```

*TyCalc> intValue one
1
*TyCalc> intValue (typeAdd one one)
2
*TyCalc>

```

Exercise 2 The inverse of reflection is reification. Define a type class *Reflect* with a method *reflect* which maps an integer to its type-level encoding.

Exercise 3 One problem with the above encoding of naturals is that the size of the type structure is linear in the size of the encoded integer. Develop a different formulation in which the encoding is logarithmic in respect to the size of the encoded integer. Extend the new formulation further by making it encode integers, not just naturals.

3 Calculating with Types and Lists

The above examples provide an introduction to calculating at the type level. The benefits, however, are difficult to determine, because everything we did at the type level could be done easier at the term level.

In this section, we show a application of type-level programming which allows us to statically guarantee operations on lists.

Instead of using the standard Haskell list datatype, we define our own lists. This is for two reasons. First, we're doing type-level calculations, so we need two type-level values. The standard list constructors, `(:)` and `[]`, close the list type. We introduce two *types*, `Nil` and `Cons`, to build our lists. Secondly, we want our list values to carry their lengths with them; we use some Haskell trickery to accomplish this.

4 The list type(s)

The `Nil` type constructor is similar to the `[]` list constructor. The type parameter is necessary because the empty list does contain a type for its elements, it's just that it is universally quantified. When something is added to the empty list type, the type parameter is unified, and fixed.

```
data Nil a = Nil deriving Show
```

```
*TyCalc> :type []
[] :: forall a. [a]
*TyCalc> :type (:)
(:) :: forall a. a -> [a] -> [a]
*TyCalc> :type (:) 'a' []
(:) 'a' [] :: [Char]
```

The `Cons` type constructor requires more explanation. First, the `n` type parameter is going to be used to encode the length of the list. This is where calculating with naturals is useful, because we can only apply type constructors to types, not terms. However, we have the mechanism to encode list lengths (*Ints*) as types; this gives us the necessary machinery. Additionally, `n` is called a *phantom type*, because it doesn't appear on the right-hand side of the equation.

The second type parameter, `a`, is the type of the list elements. Finally, the third type parameter, `l`, is the type of the tail of the `Cons` cell.

```
data Cons n a l = Cons a l deriving Show
```

Our goal is to eventually only allow homogeneous lists, where doing a `Cons` at the front results in a list one element longer. However, as it stands now, the types we have defined are even more relaxed than the standard list datatype. Not only can we build heterogeneous lists, but we can even build lists where the tail is not a list!

```
*TyCalc> Cons 1 (Cons 'a' Nil)
Cons 1 (Cons 'a' Nil)
*TyCalc> Cons 1 1
Cons 1 1
```

4.1 Adding length constraints

We want a way to extract the length of a list, as a type. This is defined using the `LLength` class. Although we're only targeting this for lists, it is conceivable that we could use the same class for other container types.

```
class LLength l n | l -> n where
  length :: l -> n
```

The length of a `Nil` list is always 0. It is because of this that we avoided adding a length phantom type parameter to the `Nil` type. The `Cons` instance declaration enforces that the length of a list resulting from `Cons`ing on a new element is one greater than the length of the tail.

```
instance LLength (Nil n) Z where
  length _ = ( $\perp$  :: Z)
instance LLength l tlength => LLength (Cons (S tlength) a l) (S tlength) where
  length _ = ( $\perp$  :: S tlength)
```

4.2 Making the list homogeneous

To insure that the list has homogeneously typed elements, we use another class to extract the elements' type. The instance declarations for *Cons* insures that *ContainerType* is only defined over homogeneous lists, due to the *ContainerType l a* constraint.

```
class ContainerType c ce | c → ce
instance ContainerType (Nil a) a
instance ContainerType l a ⇒ ContainerType (Cons n a l) a
```

Finally, we define "smart constructors" for creating values of the new type. It's necessary to use explicit type annotations when defining these constructors.

```
nil :: Nil a
nil = Nil

cons :: (ContainerType l a, LLength l n) ⇒ a → l → Cons (S n) a l
cons hd tl = Cons hd tl
```

Using these functions, we can create lists, and the type system will automatically calculate the list's length.

```
*TyCalc> :type cons 'a' (cons 'b' nil)
cons 'a' (cons 'b' nil) :: Cons
                                (S (S Z)) Char (Cons (S Z) Char (Nil Char))

*TyCalc> cons 'a' (cons 'b' nil)
Cons 'a' (Cons 'b' Nil)
```

4.3 A length-correct append

Finally, we're going to use the list types that we've created to insure the invariant that *the length of the concatenation of two lists is the sum of the lengths of the two component lists*.

Once again, we define a typeclass with a functional dependency. *Append* can be read as "*b* *tsAppended* to *a* gives the list *c*". The implementation is similar to the process used to translate from the term-level add or multiply to the type level. The term-level definition of *append* omitted, but is straightforward to implement.

```
class Append a b c | a b → c where
  tsAppend :: a → b → c
```

Appending a *Nil* list to any other list is an identity operation.

```
instance ContainerType l a ⇒ Append (Nil a) l l
  where tsAppend _ l = l
```

The *tsAppend* is defined recursively. The second list is appended to the tail of the first list, then the head of the first list is *Consed* to the result. We use the *LLength* constraints to insure the invariant property proposed earlier.

```
instance (Append l1 l2 l3, LLength l3 n) ⇒
  Append (Cons b a l1) l2 (Cons (S n) a l3) where
  tsAppend (Cons a l1) l2 = Cons a (tsAppend l1 l2)
```

We can show how the invariant holds with a simple example. We begin with an incorrect *Cons* where the length should be (S Z), not Z.

```
let x = (⊥ :: Cons Z Int (Nil Int))
```

```
*TyCalc> :type tsAppend x x

No instance for (LLength (Cons Z Int (Nil Int)) n)
  arising from use of 'tsAppend' at <No locn>
```

The *tsAppend* fails, because we have the wrong length annotation on the list. But with the correct length *Cons*, we get the correct answer.

```
*TyCalc> let x = (undefined :: Cons (S Z) Int (Nil Int))
*TyCalc> :type tsAppend x x
tsAppend x x :: Cons (S (S Z)) Int (Cons (S Z) Int (Nil Int))
```

Note that if we had use the *cons* and *nil* functions to create the lists, it wouldn't even be possible to create the values with the incorrect length annotations.

5 Conclusion

This tutorial has demonstrated some of the uses for calculating at the type level. We were able to duplicate, at the type level, the calculations available at the term level. Additionally, using our encoding we were able to use the type system to enforce some type constraints which would ordinarily require dependent types.

Exercise 4 *Create a new datatype of order-sorted lists. Define an ordering predicate over types, then define class constraints to insure that any insertion into a list maintains the list's ordering.*

References

- [1] M. P. Jones. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 230–244. Springer-Verlag, 2000.
- [2] C. McBride. Faking it: Simulating dependent types in haskell. *Journal of Functional Programming*, 12(4 and 5):375–392, July 2002.