Typed type-level programming in Haskell, part III: I can haz typs plz?

Posted on July 19, 2010

In Part II, I showed how *type families* can be used to do type-level programming in a functional style. For example, here is addition of natural numbers again:

```
data Z
data S n

type family Plus m n :: *
type instance Plus Z n = n
type instance Plus (S m) n = S (Plus m n)
```

Now, *why* might we want to do such a thing? One example (I know, I know, this is *always* the example… but hey, it's a good example) is if we wanted to have a type of polymorphic *length-indexed vectors* (or as they are sometimes known, "Length-Observed Lists") where the type of a vector includes its length. Using a generalized algebraic data type (GADT), we can write something like this:

```
data LOL :: * -> * -> * where
  KThxBye :: LOL Z a
  Moar    :: a -> LOL n a -> LOL (S n) a
```

This says that

1. `LOL` is a type constructor of kind `* -> * -> *`, that is, it takes two type arguments of kind `*` and produces a type of kind `*`. The intention is that the first argument records the length, and the second records the type of the elements.
2. `KThxBye` constructs a vector of length zero.
3. Given an element of type `a` and a vector of as of length `n`, `Moar` constructors a vector of length `S n`.

The type-level function `Plus` comes in when we implement an `append` function for our length-indexed vectors: in order to express the type of `append` we have to add the lengths of the input vectors.

```
append :: LOL m a -> LOL n a -> LOL (Plus m n) a
append KThxBye     v = v
append (Moar x xs) v = Moar x (append xs v)
```

If you haven't already seen things like this, it's a good exercise to figure out why this definition of

Follow

`append` typechecks (and why it *wouldn't* typecheck if we put anything other than `Plus m n` as the length of the output).

OK, great! We can make GHC check the lengths of our lists at compile time. So what's the problem? Well, there are (at least) three obvious things which this code leaves to be desired:

1. It doesn't matter whether we have already declared a `Nat` type with constructors `Z` and `S`; we have to redeclare some empty types `Z` and `S` to represent our type-level natural number "values". And declaring empty types to use like "values" seems silly anyway.
2. It also doesn't matter whether we've already implemented a `plus` function for our `Nat` values; we must re-code the addition algorithm at the type level with the type family `Plus`. Especially irksome is the fact that these definitions will be virtually identical.
3. Finally, and most insidiously, `LOL` is essentially *untyped*. Look again at the kind of `LOL :: * -> * -> *`. There's nothing in the kind of `LOL` that tells us the first argument is supposed to be a type-level number. Nothing prevents us from accidentally writing the type `LOL Int (S Z)` — we'll only run into (potentially confusing) problems later when we try to write down a value with this type.

Wouldn't it be nice if we could reuse (1) values and (2) functions at the type level, and (3) get more informative kinds in the bargain? Indeed, inspired by Conor McBride's SHE, our work aims precisely to enable (1) and (3) in GHC as a start, and hopefully eventually (2) (and other features) as well. Hopefully soon, you'll be able to write this:

```
data Nat = Z | S Nat

type family Plus (m::Nat) (n::Nat) :: Nat
type instance Plus Z n = n
type instance Plus (S m) n = S (Plus m n)

data LOL :: Nat -> * -> * where
  KThxBye :: LOL Z a
  Moar    :: a -> LOL n a -> LOL (S n) a

append :: ...  -- exactly the same as before
```

...or even this:

```
data Nat = Z | S Nat

plus :: Nat -> Nat -> Nat
plus Z n = n
plus (S m) n = S (plus m n)

data LOL :: Nat -> * -> * where ... -- same as above

append :: LOL m a -> LOL n a -> LOL (plus m n) a
append = ...  -- same as before
```

Follow

In another post I'll explain what the above fantasy code would be doing in a bit more detail, talk about precisely how we propose to accomplish this, and discuss why we might want to do things this way, rather than introducing full dependent types (or just chucking Haskell and all moving to Agda).

Share this:

Email Print Reddit Google Twitter Facebook More

Related

Typed type-level programming in Haskell, part II: type families
In "haskell"

Typed type-level programming in Haskell, part I: functional dependencies
In "haskell"

Typed type-level programming in Haskell, part IV: collapsing types and kinds
In "haskell"

About Brent
Assistant Professor of Computer Science at Hendrix College. Functional programmer, mathematician, teacher, pianist, follower of Jesus.
View all posts by Brent →

This entry was posted in haskell and tagged functional programming, haskell, type-level. Bookmark the permalink.

15 Responses to *Typed type-level programming in Haskell, part III: I can haz typs plz?*

Gabor *says:*
July 19, 2010 at 11:50 am

In http://code.google.com/p/omega/wiki/AutoLevelled I suggest the opposite approach to Conor's SHE: Define types (kinds, whatever) along with their type functions at the highest possible level and then push them down into the value world. I believe this approach is nicer, because you start at the well-behaved level of types and do not lose properties like termination etc., while descending, so you get isomorphisms between the levels without any effort.

Reply

Brent *says:*
July 19, 2010 at 11:57 am

Hi Gabor, thanks for the comment. I've thought a bit about this approach too. In practice I wonder how much difference it makes — it seems like it ultimately comes down to just being a question of what syntactic sugar is available in the surface language. I.e. you could be allowed to write value-level things and use them at the type level, or write type-level things and use them at the value level, but either way the code generated behind the scenes is identical. However I'm certainly open to being convinced that it does make a practical difference.

Reply

Zaphod *says:*
May 6, 2012 at 4:14 am

Follow

I'm reading this blog entry almost two years after it was written, and I'm only now learning about dependent typing related things. That puts me in a nice freshman position to comment on this severely outdated issue, as follows: Gabor's suggestion makes so much sense! It's simple and obvious design. I'm just about to proceed to read about ghc 7.6 and find out how things ended up. Exciting! :)

Reply

zygoloid *says:*
July 19, 2010 at 2:32 pm

Hi Brent,

If you want to lift value-level functions to the type level, I imagine you immediately hit upon the problem that Uppercase means something different in the two worlds: Value-level "plus" is lowercase but type-level "Plus" would need to be uppercase so the type system knows what to implicitly quantify over (not a concern at the value level since each variable must be introduced explicitly). Conversely, at the value level, we need to capitalize constructors so we can distinguish "let Just x = 0 in True" from "let just x = 0 in True" (not a concern at the type level since we don't use the same syntax for defining type functions and pattern matching).

Do you have a solution to this problem? Automatic capitalization seems like a substandard choice: it would make it harder to search for the definition of an identifier, and prepending : makes type operators look ugly. Do we have to give up automatic implicit quantification of type variables? For type operators that seems reasonable, but for the non-operator case it sounds like a significant loss. Perhaps it won't be so bad if we only need explicit quantification for type signatures including an implicitly-lifted type function.

Reply

Brent *says:*
July 20, 2010 at 2:26 am

I'm not sure I quite understand the issue. Can you give an example of the sort of problem you are imagining?

Reply

Austin Seipp *says:*
July 20, 2010 at 12:48 pm

Hi Brent,

I think he's talking about this example in particular:

```
append :: LOL m a -> LOL n a -> LOL (plus m n) a
append = ...
```

In this case, the value-function 'plus' which has been lifted to the type-level is a lowercase identifier in this type – the distinction is important because a lower-case identifier in a type counts as a type variable that GHC is free to quantify over, does it not? This seems like a reasonable problem but not an insurmountable one.

Reply

Follow

José Pedro Magalhães *says:*
July 20, 2010 at 2:38 am

Very much looking forward to this…

Reply

Austin Seipp *says:*
July 20, 2010 at 12:54 pm

Hi Brent,

Is there any publicly viewable branch of the current work you and the GHC guys are doing? It would be neat to toy with stuff like this if it's possible at this point. I'm also wondering how this relates to the newest work by SPJ & Co. on their OutsideIn(X) system if they're even related – I would assume from the examples here that a lot of the type inference here may be limited in cases like this.

And for that matter, what about non-total value-level functions raised to the type level? I'm surprised nobody has mentioned it this far!

Maybe I should just wait for the part 4…

Reply

Edward Kmett *says:*
July 22, 2010 at 2:48 pm

Gah, the anticipation is killing me! How dare you spend all your time actually implementing this stuff rather than blogging about it!

Reply

Brent *says:*
July 22, 2010 at 3:12 pm

Hehehe. Current plan: (1) revise Haskell Symposium paper (today + tomorrow) (2) visit the Netherlands (Sat-Mon) (3) write blog post to appease Edward (Tues)

Reply

José Pedro Magalhães *says:*
July 22, 2010 at 3:14 pm

Visiting the NL? Planning to visit Utrecht University too?

Reply

Brent *says:*
July 22, 2010 at 3:18 pm

That would be a lot of fun, but probably not this weekend — we'll be visiting a friend in Leiden.

Reply

Follow

Pingback: *Typed type-level programming in Haskell, part IV: collapsing types and kinds « blog :: Brent -> [String]*

Vivian McPhail *says:*
September 5, 2010 at 7:28 pm

Hi,

This comment is less about types as values, values as types, than as the purported application domain.

I agree that length-observed lists ‘work,’ in the sense that the type-checker can tell me that I am attempting to add lists of different lengths. However, when does this help? Surely it helps when I am playing with a toy problem at the REPL prompt, where compilation/execution is a single interpretation pass. But, what about when I load two vectors from file in an executable? Suppose I have one vector a, length 1000, and another b, length 1001. Type erasure means that there is no typechecking at runtime (modulo my attempt at dynamic typing with GHC API).

Even if we decide to dynamically invoke a type-checker at runtime and find that the types of vectors a and b (which we loaded after execution commenced) don’t match and therefore cannot be added, I put forward that a simple _value-level_ check (| length a /= length b = throwError “incompatible vector length”) is going to be _computationally_ more efficient.

Remember, if we want LOL at runtime of dynamically loaded vectors, we somehow have to invoke type-checking at dynamic load time (post-compilation, post-execution), but type-checking is a difficult theorem-proving task which is optimised not for computational efficiency, but in other respects.

In any case, any sort of type checking magic is going to end up somewhere as a (length a /= length b) test, whether magically inserted by a typechecker or manually inserted by an optimising programmer.

Cheers!

Reply

Pingback: *Typed type-level programming: status report « blog :: Brent -> [String]*

Follow