blog :: Brent -> [String]

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

Posted on July 6, 2010

In my previous post, we saw how multi-parameter type classes with functional dependencies in Haskell allow us to do type-level programming in a logic programming style. (If you're not clear on why this corresponds to a logic programming style, see the ensuing discussion on reddit, where others explained it much better than I did in my post.)

However, MPTCs + FDs weren't the last word on type-level programming. In 2007, along came type families.

Essentially, type families allow us to write *functions on types*. For example, here's how we would implement the same `Plus` function from the last post, this time using type families:

```
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)
```

This says that for any types `m` and `n`, `Plus m n` is type of kind `*`. But it isn't a *new* type, it's just an alias for some existing type. It's instructive to think carefully about the difference between this and type synonyms. After all, using a type synonym declaration, we can already make `Plus m n` an alias for some existing type, right?

Well, yes, but the difference is that a type synonym *doesn't get to look at its arguments*. The technical term for this is that type synonyms must be *parametric*. So, for example, we can say

```
type Foo m n = [(m, Maybe n)]
```

which defines the type synonym `Foo` uniformly for all arguments `m` and `n`, but using only type synonyms we *cannot* say

```
type Foo m Int = [m]
type Foo m Char = Maybe m
```

where `Foo` acts differently depending on what its second argument is. However, this is precisely what type families allow us to do — to declare type synonyms that do pattern-matching on type arguments. Looking back at the `Plus` example above, we can see that it evaluates to different types depending on whether its first argument is `Z` or `S n`. Notice also that it is essentially identical to the way we would implement addition on regular value-level natural numbers, using pattern-matching on the first argument and a recursive call in the successor case:

```
data Nat = Z | S Nat

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

Let’s check that `Plus` works as advertised:

```
*Main> :t undefined :: Plus (S Z) (S Z)
undefined :: Plus (S Z) (S Z) :: Plus (S Z) (S Z)
```

Well, unfortunately, as a minor technical point, we can see from the above that ghci doesn’t expand the type family for us. The only way I currently know how to force it to expand the type family is to generate a suitable error message:

```
*Main> undefined :: Plus (S Z) (S Z)

...No instance for (Show (S (S Z)))...
```

This is ugly, but it works: `S (S Z)` is the reduced form of `Plus (S Z) (S Z)`.

So type families let us program in a *functional* style. This is nice — I daresay most Haskell programmers will be more comfortable only having to use a single coding style for both the value level and the type level. There are a few cases where a logic programming style can be quite convenient (for example, with an additional functional dependency we can use the `Plus` type class from the last post to compute both addition *and* subtraction), but in my opinion, the functional style is a huge win in most cases. (And, don’t worry, FDs and TFs are equivalent in expressiveness.)

Of course, there is a lot more to all of this; for example, I haven’t even mentioned data families or associated types. For more, I recommend reading the excellent tutorial by Oleg Kiselyov, Ken Shan, and Simon Peyton Jones, or the page on the GHC wiki. For full technical details, you can look at the System FC paper.

Nothing is ever perfect, though — in my next post, I’ll explain what type families still leave to be desired, and what we’re doing to improve things.

Share this:

Email | Print | Reddit | Google | Twitter | Facebook | More

Related

Typed type-level programming in Haskell, part III: I can haz typs plz?
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 programming, type families, type-level, types. Bookmark the permalink.

5 Responses to *Typed type-level programming in Haskell, part II: type families*

Andrea Vezzosi *says:*
July 6, 2010 at 10:10 am

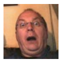
looking forward to your next post, especially if there'll be news on closed type families :)

Reply

Brent *says:*
July 6, 2010 at 5:53 pm

You might just be in luck, we shall see! =)

Reply



augustss *says:*
July 8, 2010 at 4:09 pm

A Haskell 'type' declaration is like a positive function space and a 'type family' is like a negative function space, using the terminology of Harper&al focus logic.

Reply

Pingback: *Typed type-level programming in Haskell, part III: I can haz typs plz? « blog :: Brent -> [String]*

Pingback: *Functions don't just have types: They ARE Types. And Kinds. And Sorts. Help put a blown mind back together | PHP Developer Resource*

Follow