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

Posted on June 29, 2010

The other project I'm working on at MSR this summer is a bit more ambitious: our headline goal is to extend GHC to enable *typed, functional, type-level programming*. What's that, you ask? Well, first, let me tell you a little story…

Once upon a time there was a lazy*, pure, functional programming language called Haskell. It was very careful to always keep its values and types strictly separated. So of course "type-level programming" was completely out of the question! …or was it?

In 1997, along came *multi-parameter type classes*, soon followed by functional dependencies. Suddenly, type-level programming became possible (and even fun and profitable, depending on your point of view). How did this work?

Whereas normal type classes represent *predicates* on types (each type is either an instance of a type class or it isn't), multi-parameter type classes represent *relations* on types. For example, if we create some types to represent natural numbers,

```
data Z
data S n
```

we can define a multi-parameter type class `Plus` which encodes the addition relation on natural numbers:

```
class Plus m n r

instance Plus Z n n
instance (Plus m n r) => Plus (S m) n (S r)
```

This says that for any types `m`, `n`, and `r`, `(Z,n,n)` are in the `Plus` relation, and `(S m, n, S r)` are in the `Plus` relation whenever `(m,n,r)` are. We can load this into `ghci` (after enabling a few extensions, namely `MultiParamTypeClasses`, `FlexibleInstances`, and `EmptyDataDecls`), but unfortunately we can't yet actually use the `Plus` relation to do any type-level *computation*:

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

We asked for the type of something which has type `r`, given that the relation `Plus (S Z) (S Z)`

— but notice that ghci was not willing to simplify that constraint at all. The reason is that type classes are *open* — there could be lots of instances of the form `Plus (S Z) (S Z) r` for many different types `r`, and `ghci` has no idea which one we might want.

To the rescue come functional dependencies, which let us specify that some type class parameters are determined by others — in other words, that the relation determined by a multi-parameter type class is actually a *function*.

```
class Plus m n r | m n -> r
instance Plus Z n n
instance (Plus m n r) => Plus (S m) n (S r)
```

Here we've added the functional dependency `m n -> r`, which says that knowing `m` and `n` also determines `r`. In practice, this means that we are only allowed to have a single instance of `Plus` for any particular combination of `m` and `n`, and if ghc can determine `m` and `n` and finds an instance matching them, it will assume it is the only one and pick `r` to match. Now we can actually do some computation (after enabling `UndecidableInstances`):

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

Aha! So 1 + 1 = 2, at the type level!

Type-level programming with multi-parameter type classes and functional dependencies has a strong resemblance to [logic programming](#) in languages like Prolog. We declare rules defining a number of relations, and then "running" a program consists of searching through the rules to find solutions for unconstrained variables in a given relation. (The one major difference is that GHC's type class instance search doesn't (yet?) do any backtracking.)

This is getting a bit long so I'll break it up into a few posts. In the next installment, I'll explain type families, which are a newer, alternative mechanism for type-level programming in Haskell.

* OK, OK, non-strict.

Related

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

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

Back from Baltimore
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 addition, functional dependencies, programming, type-level. Bookmark the permalink.

13 Responses to *Typed type-level programming in Haskell, part I: functional dependencies*

Edward Z. Yang *says:*
June 29, 2010 at 9:08 am

I've been doing some thinking about this (while waiting in the long lines at the World Expo), and one of the things that I'm a little concerned about is making the errors for when things go wrong in type level programming comprehensible (which have a reputation—though I haven't verified myself—for being helplessly arcane), so I'd love to see if you guys have been thinking about this problem. :-)

Reply

Brent *says:*
June 29, 2010 at 9:30 am

Yes, error messages when doing type-level programming stuff can be quite hairy. It's a very important problem, and one I admit we haven't thought about too much at this point. But I definitely intend to think about it more at some point, once we have a working implementation. =)

Reply

beroal *says:*
June 29, 2010 at 10:11 pm

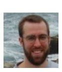
I second that. It will be a great day when instance inference (I prefer this term to "type inference" when talking about Haskell type classes) become a real programming language. Because "programming language" means debugging, formal semantics, extensions.

Reply

Ivan Lazar Miljenovic *says:*
June 29, 2010 at 10:39 pm

I'm confused; what's the point of the `r' type parameter to the typeclass?

Reply



Brent *says:*
June 30, 2010 at 2:01 am

The typeclass represents the addition *relation*, which is a three-place relation. For example, (3,5,8) are in the addition relation, because 3 + 5 = 8. If there is an instance `Plus m n r`, then m + n = r.

Reply

[Ivan Lazar Miljenovic] *says:*

June 30, 2010 at 2:08 am

OK, with that explanation your code even makes more sense.

Reply

[Bartosz Milewski] *says:*

June 30, 2010 at 7:23 pm

What line of code generated this output?
undefined :: (Plus (S Z) (S Z) r) => r :: S (S Z)

Reply

Brent *says:*

July 1, 2010 at 3:35 am

That is ghci's response to the :t command right above it. It is informing us that the type of `(undefined :: (Plus (S Z) (S Z) r) => r)` is `S (S Z)`. It's a bit convoluted but this is how you can get ghci to do type-level computation, by asking for the type of undefined with a type annotation.

Reply

Pingback: *Typed type-level programming in Haskell, part II: type families « blog :: Brent -> [String]*

Brian *says:*

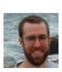
August 7, 2010 at 8:58 am

I followed the example (http://hpaste.org/fastcgi/hpaste.fcgi/view?id=28672), but when I tried to run ":t undefined :: (Plus (S Z) (S Z) r) => r" command on ghci, I get the following error msg:

Non type-variable argument in the constraint: Plus (S Z) (S Z) r
(Use -XFlexibleContexts to permit this)
In an expression type signature: (Plus (S Z) (S Z) r) => r
In the expression: undefined :: (Plus (S Z) (S Z) r) => r

Why does it complain about FlexibleContexts when I already enabled that pragma? Does this example not work on ghc 6.12.1?

Reply



Brent *says:*

August 7, 2010 at 11:06 am

The LANGUAGE pragma only applies to the file itself; you need to enable the pragma separately in ghci to be able to use it in expressions that you type at the prompt. So either pass ghci the -XFlexibleInstances flag when you start it, or once ghci is running you can :set -XFlexibleInstances.

Reply

[Jim Stuttard] *says:*

May 26, 2013 at 2:50 am

ghc-7.6.3 needs :set -XFlexibleContexts as well.

Reply

Pingback: *Links and Activities, 9/2015 | Mental Wilderness*
