What's wrong with GHC Haskell's current constraint system?

I've heard that there are some problems with Haskell's "broken" constraint system, as of GHC 7.6 and below. What's "wrong" with it? Is there a comparable existing system that overcomes those flaws?

For example, both edwardk and tekmo have run into trouble (e.g. this comment from tekmo).

haskell typeclass

edited Jan 15 '13 at 5:07

tobyodavies
10.3k ● 2 ● 26 ● 48

asked Oct 9 '12 at 17:34

Dan Burton
31.3k ● 13 ● 85 ● 159

4 While I'm sure there's an interesting question in here, in its current form it's essentially "What problems have edwardk and tekmo run into?" , which can only really be answered by those people. As such, I don't think this question is a good fit for SO in its current form. – hammar Oct 9 '12 at 17:47

5 I seems like "what problems exist that anyone has run into?" is more the intent here. Anyone who's run into similar problems could, I expect, recognize that and field the question just as well as the specific people whose complaints are mentioned. – C. A. McCann Oct 9 '12 at 18:22

3 Yes, @C.A.McCann captured my intent fairly well, though I'm not particularly looking for "what problems have you run into?" so much as "what is the underlying problem?" I expect a good answer will elaborate on what the current constraint system *is*, what its weaknesses are, and whether there are existing plans to improve on it. – Dan Burton Oct 9 '12 at 18:56

6 I started a discussion at /r/haskell. I was under the impression that there was an obvious, well-understood flaw, but apparently this is not the case. – Dan Burton Oct 9 '12 at 21:07

2 @C.A.McCann what is LtU? – Cetin Sert Oct 10 '12 at 13:40

2 Answers

Ok, I had several discussions with other people before posting here because I wanted to get this right. They all showed me that all the problems I described boil down to the lack of polymorphic constraints.

The simplest example of this problem is the `MonadPlus` class, defined as:

```
class MonadPlus m where
    mzero :: m a
    mplus :: m a -> m a -> m a
```

... with the following laws:

```
mzero `mplus` m = m

m `mplus` mzero = m

(m1 `mplus` m2) `mplus` m3 = m1 `mplus` (m2 `mplus` m3)
```

Notice that these are the `Monoid` laws, where the `Monoid` class is given by:

```
class Monoid a where
    mempty :: a
    mappend :: a -> a -> a

mempty `mplus` a = a

a `mplus` mempty = a

(a1 `mplus` a2) `mplus` a3 = a1 `mplus` (a2 `mplus` a3)
```

So why do we even have the `MonadPlus` class? The reason is because Haskell forbids us from writing constraints of the form:

```
(forall a . Monoid (m a)) => ...
```

So Haskell programmers must work around this flaw of the type system by defining a separate class to handle this particular polymorphic case.

However, this isn't always a viable solution. For example, in my own work on the `pipes` library, I frequently encountered the need to pose constraints of the form:

```
(forall a' a b' b . Monad (p a a' b' b m)) => ...
```

Unlike the `MonadPlus` solution, I cannot afford to switch the `Monad` type class to a different type class to get around the polymorphic constraint problem because then users of my library would lose `do` notation, which is a high price to pay.

This also comes up when composing transformers, both monad transformers and the proxy transformers I include in my library. We'd like to write something like:

```
data Compose t1 t2 m r = C (t1 (t2 m) r)

instance (MonadTrans t1, MonadTrans t2) => MonadTrans (Compose t1 t2) where
    lift = C . lift . lift
```

This first attempt doesn't work because `lift` does not constrain its result to be a `Monad`. We'd actually need:

```
class (forall m . Monad m => Monad (t m)) => MonadTrans t where
    lift :: (Monad m) => m r -> t m r
```

... but Haskell's constraint system does not permit that.

This problem will grow more and more pronounced as Haskell users move on to type constructors of higher kinds. You will typically have a type class of the form:

```
class SomeClass someHigherKindedTypeConstructor where
    ...
```

... but you will want to constrain some lower-kinded derived type constructor:

```
class (SomeConstraint (someHigherKindedTypeConstructor a b c))
    => SomeClass someHigherKindedTypeConstructor where
    ...
```

However, without polymorphic constraints, that constraint is not legal. I've been the one complaining about this problem the most recently because my `pipes` library uses types of very high kinds, so I run into this problem constantly.

There are workarounds using data types that several people have proposed to me, but I haven't (yet) had the time to evaluate them to understand which extensions they require or which one solves my problem correctly. Somebody more familiar with this issue could perhaps provide a separate answer detailing the solution to this and why it works.

edited Oct 11 '12 at 20:16

answered Oct 11 '12 at 15:29

Gabriel Gonzalez

27.1k ● 3 ● 48 ● 99

[a follow-up to Gabriel Gonzalez answer]

The right notation for constraints and quantifications in Haskell is the following:

```
<functions-definition> ::= <functions> :: <quantified-type-expression>

<quantified-type-expression> ::= forall <type-variables-with-kinds> . (<constraints>) =>
<type-expression>

<type-expression> ::= <type-expression> -> <quantified-type-expression>
                    | ...

...
```

Kinds can be omitted, as well as `forall`s for rank-1 types:

```
<simply-quantified-type-expression> ::= (<constraints-that-uses-rank-1-type-variables>) =>
<type-expression>
```

For example:

```
{-# LANGUAGE Rank2Types #-}

msum :: forall m a. Monoid (m a) => [m a] -> m a
msum = mconcat

mfilter :: forall m a. (Monad m, Monoid (m a)) => (a -> Bool) -> m a -> m a
```

```
mfilter p ma = do { a <- ma; if p a then return a else mempty }

guard :: forall m. (Monad m, Monoid (m ())) => Bool -> m ()
guard True = return ()
guard False = mempty
```

or without `Rank2Types` (since we only have rank-1 types here), and using `CPP` (j4f):

```
{-# LANGUAGE CPP #-}

#define MonadPlus(m, a) (Monad m, Monoid (m a))

msum :: MonadPlus(m, a) => [m a] -> m a
msum = mconcat

mfilter :: MonadPlus(m, a) => (a -> Bool) -> m a -> m a
mfilter p ma = do { a <- ma; if p a then return a else mempty }

guard :: MonadPlus(m, ()) => Bool -> m ()
guard True = return ()
guard False = mempty
```

The "problem" is that we can't write

```
class (Monad m, Monoid (m a)) => MonadPlus m where
  ...
```

or

```
class forall m a. (Monad m, Monoid (m a)) => MonadPlus m where
  ...
```

That is, `forall m a. (Monad m, Monoid (m a))` can be used as a standalone constraint, but can't be aliased with a new one-parametric typeclass for `*->*` types.

This is because the typeclass defintion mechanism works like this:

```
class (constraints[a, b, c, d, e, ...]) => ClassName (a b c) (d e) ...
```

i.e. the **rhs** side introduce type variables, not the lhs or `forall` at the lhs.

Instead, we need to write 2-parametric typeclass:

```
{-# LANGUAGE MultiParamTypeClasses, FlexibleContexts, FlexibleInstances #-}

class (Monad m, Monoid (m a)) => MonadPlus m a where
  mzero :: m a
  mzero = mempty
  mplus :: m a -> m a -> m a
  mplus = mappend

instance MonadPlus [] a
instance Monoid a => MonadPlus Maybe a

msum :: MonadPlus m a => [m a] -> m a
msum = mconcat

mfilter :: MonadPlus m a => (a -> Bool) -> m a -> m a
mfilter p ma = do { a <- ma; if p a then return a else mzero }

guard :: MonadPlus m () => Bool -> m ()
guard True = return ()
guard False = mzero
```

Cons: we need to specify second parameter every time we use `MonadPlus`.

Question: how

```
instance Monoid a => MonadPlus Maybe a
```

can be written if `MonadPlus` is one-parametric typeclass? `MonadPlus Maybe` from `base`:

```
instance MonadPlus Maybe where
   mzero = Nothing
   Nothing `mplus` ys  = ys
   xs      `mplus` _ys = xs
```

works not like `Monoid Maybe`:

```
instance Monoid a => Monoid (Maybe a) where
  mempty = Nothing
  Nothing `mappend` m = m
  m `mappend` Nothing = m
  Just m1 `mappend` Just m2 = Just (m1 `mappend` m2) -- < here
```

:

```
(Just [1,2] `mplus` Just [3,4]) `mplus` Just [5,6] => Just [1,2]
(Just [1,2] `mappend` Just [3,4]) `mappend` Just [5,6] => Just [1,2,3,4,5,6]
```

Analogically, `forall m a b n c d e. (Foo (m a b), Bar (n c d) e)` gives rise for (7 - 2 * 2)-parametric typeclass if we want `*` types, (7 - 2 * 1)-parametric typeclass for `* -> *` types, and (7 - 2 * 0) for `* -> * -> *` types.

answered Oct 11 '12 at 19:43

JJJ
1,756 ● 4 ● 18