Haskell Symposium 2014

Gothenburg, Sweden

Promoting Functions to Type Families in Haskell

Richard A. Eisenberg
University of Pennsylvania

Jan Stolarek
Politechnika Łódzka

Support for type-level programming in GHC

GHC provides many extensions for type-level programming:

- functional dependencies
- GADTs
- open type families
- datakinds
- kind polymorphism
- type-level literals
- closed type families

Are we there yet?

Support for type-level programming in GHC

A lot of constructs are missing at the type level:

- lambdas
- partial application
- higher order functions
- `case` expressions
- `let` statements
- `where` clauses
- guards
- typeclasses
- records
- arithmetic sequences
- infinite data structures
- higher-kinded types
- do-notation
- list comprehensions

Our answer is "almost"

Towards dependently-typed Haskell

Identify a subset of term language that can be encoded at the type level.

Adam Gundry's work[1] reformulates Core. Ours does not.

Adam Gundry's shared subset excludes partial application at the type level. Ours does not.

A step towards dependently-typed Haskell.

[1] *Type Inference, Haskell and Dependent Types*, Gundry 2013

Our contributions

Provide programmers with access to convenient type-level programming by promoting term-level definitions to the type level.

Our solution implemented using Template Haskell. Available as `singletons` library.

Explore the design space for a possible future GHC extension.

Key ideas behind promotion

- use lambda lifting (Johnson, 1985) to promote case, let and lambdas
- use defunctionalization (Reynolds, 1972) to implement partial application and first-class functions at the type level

Promoting `case`, `let` and lambdas with lambda lifting

To promote `case`, `let` and lambdas we:

- convert each to a type family
- in-scope bindings become explicit parameters

Promoting case expression - an example

Term-level definition:

```
fromMaybe :: a -> Maybe a -> a
fromMaybe d x = case x of
  Nothing -> d
  Just v  -> v
```

Promoted type-level definitions:

```
type family FromMaybe (t1 :: a) (t2 :: Maybe a) :: a
    where
      FromMaybe d x = Case d x x


type family Case d x scrut where
  Case d x Nothing  = d
  Case d x (Just v) = v
```

Why do we need defunctionalization?

GHC's type inference assumptions

GHC's type inference relies crucially on these assumptions to perform type decomposition:

1. `(a b) ~ (a c)` implies `b ~ c` (injectivity)
2. `(a b) ~ (c d)` implies `a ~ c` (generativity)

Both are true for type constructors. But neither is for type families.

Allowing type variables to unify with unsaturated type families would be incompatible with these assumptions. Defunctionalization allows us to loosen up that restriction.

Defunctionalization by example

```
data Nat = Z | S Nat
```

```
pred :: Nat -> Nat
pred Z     = Z
pred (S n) = n
```

`pred` can be used unsaturated, eg. it can be passed as an argument to a higher-order function:

```
map pred [Z, S Z]
```

Defunctionalization by example

```
data Nat = Z | S Nat

type family Pred (a :: Nat) :: Nat where
    Pred Z     = Z
    Pred (S n) = n
```

But it is invalid to write:

```
Map Pred '[Z, S Z]
```

Defunctionalization by example

```
data Nat = Z | S Nat

type family Pred (a :: Nat) :: Nat where
    Pred Z     = Z
    Pred (S n) = n
```

Instead, we will write this:

```
Map PredSym '[Z, S Z]

data PredSym :: Nat ->> Nat


PredSym @@ n = Pred n
```

Applying symbols

To use the symbols we define application operator:

```
type family (f :: k1 ->> k2) @@ (x :: k1) :: k2
```

In other words `@@` has the kind `(k1 ->> k2) -> (k1 -> k2)`: it turns our symbols into actual functions that GHC can apply.

Promotion in action

```
data Nat = Z | S Nat

$(promote [d|
  pred :: Nat -> Nat
  pred Z     = Z
  pred (S n) = n
|])
```

Promotion in action

```
$(promote [d|
  nub                          :: (Eq a) => [a] -> [a]
  nub l                        = nub’ l []
    where
      nub’ [] _                = []
      nub’ (x:xs) ls
          | x ‘elem‘ ls        = nub’ xs ls
          | otherwise          = x : nub’ xs (x:ls)
|])
```

Could we avoid defunctionalization?

Could we avoid defunctionalization?

We believe the answer is "yes".

We use symbols to work around GHC's current limitations.

But our encoding is not incompatible with GHC's type inference.

Could we avoid defunctionalization?

We only need explicit type family application `@@` and a new kind `->>` for non-injective non-generative type-level functions:

```
type family Map (f :: a ->> b) (xs :: [a]) :: [b] where
   Map f []       = []
   Map f (x : xs) = (f @@ x) : (Map f xs)
```

Unpromotable language features

Unpromotable language features

- **infinite terms**

```
iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)
```

- **arithmetic sequences** that use infinite terms (`[1..]`)
- **literals** limited by GHC's built-in literal promotion
- `Show` **and** `Read` **typeclasses** require manipulation of strings
- `do`-**notation** would require higher-sorted kinds
- **list comprehensions** are syntax sugar for the `do`-notation

Your next steps

- `cabal install singletons`
- `https://github.com/goldfirere/singletons`
- read the paper to learn about:
  - promoting `Prelude` and some `Data.*` modules
  - promoting type classes and instances
  - formal proof of our algorithm
  - kind inference
  - and more...
- start dependently-typed programming in Haskell today
- send pull requests to `https://github.com/sweirich/dth`

Haskell Symposium 2014

Gothenburg, Sweden

Promoting Functions to Type Families in Haskell

Richard A. Eisenberg
University of Pennsylvania

Jan Stolarek
Politechnika Łódzka

Encoding ->>

We declare

```
data TyFun :: * -> * -> *
```

and write `'TyFun a b -> *` to express `a ->> b`.

But we prefer

```
(a ->> b) ->> Maybe a ->> Maybe b
```

to

```
TyFun (TyFun a b -> *) (TyFun (Maybe a) (Maybe b) -> *) -> *
```

Classifying functions

GHC uses -> to classify different kinds of functions:

- **term-level functions**: can be partially applied; neither generative nor injective,
- **type constructors**: can be partially applied; both generative and injective,
- **type families**: cannot be partially applied; neither generative nor injective.

We introduce ->> to classify type-level functions that can be partially applied and are neither generative nor injective.

Defunctionalization by (a more involved) example

```
type family Plus (a :: Nat) (b :: Nat) :: Nat where
    Plus Z     m = m
    Plus (S n) m = S (Plus n m)

data PlusSym0 :: Nat ->> Nat ->> Nat
data PlusSym1 :: Nat ->  Nat ->> Nat
     Plus     :: Nat ->  Nat ->  Nat

type instance  PlusSym0    @@ n = PlusSym1 n
type instance (PlusSym1 n) @@ m = Plus n m
```