Typed type-level functional programming with GHC

Brent Yorgey
University of Pennsylvania

Haskell Implementors' Workshop
October 1, 2010

What I Did On My Summer Vacation

Brent Yorgey
University of Pennsylvania

Haskell Implementors' Workshop
October 1, 2010

What I Did On My Summer ~~Vacation~~ Holiday

Brent Yorgey
University of Pennsylvania

Haskell Implementors' Workshop
October 1, 2010

Joint work-in-progress with:

Simon Peyton-Jones

Dimitrios Vytiniotis

Stephanie Weirich

Steve Zdancewic

Outline

Type-level programming

Theory

Implementation

Future work

Type-level naturals

```
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)
```

Length-indexed vectors

```
data Vec :: * -> * -> * where
  Nil  :: Vec Z a
  Cons :: a -> Vec n a -> Vec (S n) a
```

Length-indexed vectors

```
data Vec :: * -> * -> * where
  Nil  :: Vec Z a
  Cons :: a -> Vec n a -> Vec (S n) a

append :: Vec m a -> Vec n a -> Vec (Plus m n) a
append Nil         v = v
append (Cons x xs) v = Cons x (append xs v)
```

Problems

```
data Nat = Z | S Nat

data Z     -- duplicate!
data S n
```

Problems

```
data Nat = Z | S Nat

data Z    -- duplicate!
data S n

data Vec :: * -> * -> *   -- untyped!
```

Problems

```
data Nat = Z | S Nat

data Z     -- duplicate!
data S n

data Vec :: * -> * -> *    -- untyped!

Vec Int (S Z)  -- ?
Vec (S Z) Int  -- ?
```

The goal

Taking inspiration from SHE…

The goal

Taking inspiration from ~~SHE~~ HER...

The goal

```
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)
```

The goal

```
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 Vec :: Nat -> * -> * where
  Nil  :: Vec Z a
  Cons :: a -> Vec n a -> Vec (S n) a

append :: ...
```

The goal

```
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 Vec :: Nat -> * -> * where
  Nil  :: Vec Z a
  Cons :: a -> Vec n a -> Vec (S n) a

append :: ...
```

...Look, ma, no braces!

Outline

Type-level programming

Theory

Implementation

Future work

GHC core

$$
\begin{aligned}
e ::= {} & x \mid K \\
\mid {} & \Lambda a : \kappa . e \mid e\,\tau \\
\mid {} & \lambda x : \sigma . e \mid e_1\, e_2 \\
\mid {} & \mathit{let} \ldots \mid \mathit{case} \ldots \\
\mid {} & e \rhd \gamma
\end{aligned}
$$

GHC core

$$
\begin{aligned}
e ::= {} & x \mid K \\
\mid {} & \Lambda a : \kappa . e \mid e\, \tau \\
\mid {} & \lambda x : \sigma . e \mid e_1\, e_2 \\
\mid {} & \mathit{let} \ldots \mid \mathit{case} \ldots \\
\mid {} & e \rhd \gamma
\end{aligned}
$$

$$
\begin{aligned}
\tau ::= {} & a \mid T \\
\mid {} & \tau_1\, \tau_2 \mid F_n\, \overline{\tau}^n \\
\mid {} & \forall a : \kappa . \tau
\end{aligned}
$$

GHC core

$$
\begin{aligned}
e ::= {} & x \mid K \\
\mid {} & \Lambda a : \kappa . e \mid e\,\tau \\
\mid {} & \lambda x : \sigma . e \mid e_1\, e_2 \\
\mid {} & \mathit{let} \ldots \mid \mathit{case} \ldots \\
\mid {} & e \triangleright \gamma
\end{aligned}
$$

$$
\begin{aligned}
\tau ::= {} & a \mid T \\
\mid {} & \tau_1\, \tau_2 \mid F_n\, \overline{\tau}^n \\
\mid {} & \forall a : \kappa . \tau
\end{aligned}
$$

$$
\kappa ::= \star \mid \kappa_1 \to \kappa_2
$$

GHC core

$$\begin{aligned} e ::= {} & x \mid K \\ \mid {} & \Lambda a : \kappa . e \mid e\,\tau \\ \mid {} & \lambda x : \sigma . e \mid e_1\, e_2 \\ \mid {} & let \ldots \mid case \ldots \\ \mid {} & e \triangleright \gamma \end{aligned}$$

$$\begin{aligned} \tau ::= {} & a \mid T \mid K \\ \mid {} & \tau_1\, \tau_2 \mid F_n\, \overline{\tau}^n \\ \mid {} & \forall a : \kappa . \tau \end{aligned}$$

$$\kappa ::= \star \mid \kappa_1 \to \kappa_2$$

GHC core

$$
\begin{aligned}
e ::= {} & x \mid K \\
\mid {} & \Lambda a : \kappa . e \mid e\, \tau \\
\mid {} & \lambda x : \sigma . e \mid e_1\, e_2 \\
\mid {} & \mathit{let} \ldots \mid \mathit{case} \ldots \\
\mid {} & e \triangleright \gamma
\end{aligned}
$$

$$
\begin{aligned}
\boldsymbol{\kappa} ::= {} & a \mid T \mid K \mid \star \\
\mid {} & \boldsymbol{\kappa}_1 \boldsymbol{\kappa}_2 \mid F_n \overline{\boldsymbol{\kappa}}^n \\
\mid {} & \forall a : \boldsymbol{\kappa} . \boldsymbol{\kappa}
\end{aligned}
$$

GHC core

$$
\begin{aligned}
e ::=\ & x \mid K \\
\mid\ & \Lambda a : \kappa.e \mid e\,\tau \\
\mid\ & \lambda x : \sigma.e \mid e_1\, e_2 \\
\mid\ & let\ldots \mid case\ldots \\
\mid\ & e \rhd \gamma
\end{aligned}
$$

$$
\begin{aligned}
\boldsymbol{\kappa} ::=\ & a \mid T \mid K \mid \star \\
\mid\ & \boldsymbol{\kappa}_1 \boldsymbol{\kappa}_2 \mid F_n \overline{\boldsymbol{\kappa}}^n \\
\mid\ & \forall a : \boldsymbol{\kappa}.\boldsymbol{\kappa}
\end{aligned}
$$

$$\Gamma \vdash \star : \star$$

GHC core

$$
\begin{aligned}
e ::= {} & x \mid K \\
\mid {} & \Lambda a : \kappa . e \mid e\, \tau \\
\mid {} & \lambda x : \sigma . e \mid e_1\, e_2 \\
\mid {} & \mathit{let}\ldots \mid \mathit{case}\ldots \\
\mid {} & e \rhd \gamma
\end{aligned}
\qquad
\begin{aligned}
\boldsymbol{\kappa} ::= {} & a \mid T \mid K \mid \star \\
\mid {} & \boldsymbol{\kappa}_1 \boldsymbol{\kappa}_2 \mid F_n \overline{\boldsymbol{\kappa}}^n \\
\mid {} & \forall a : \boldsymbol{\kappa} . \boldsymbol{\kappa}
\end{aligned}
$$

$$\Gamma \vdash \star : \star$$

…Why not collapse everything?

Collapse everything?

- Phase distinction!

Collapse everything?

- Phase distinction!
- No need for erasure analysis

Collapse everything?

- Phase distinction!
- No need for erasure analysis
- Incremental changes

Kind polymorphism!

$$\forall \kappa : \star . \forall a : \kappa . \ldots$$

Typechecking coercions

$$\forall a : \kappa.\tau_1 \sim \forall a : \kappa.\tau_2$$

Typechecking coercions

$$\forall a : \kappa_1 . \tau_1 \sim \forall a : \kappa_2 . \tau_2$$

Typechecking coercions

$$\forall a : \kappa_1.\tau_1 \sim \forall a : \kappa_2.\tau_2$$

- Nontrivial kind equalities only come from GADTs...

Typechecking coercions

$$\forall a : \kappa_1.\tau_1 \sim \forall a : \kappa_2.\tau_2$$

- Nontrivial kind equalities only come from GADTs...
- No lifting GADTs! (For now.)

Outline

Progress

- Currently refactoring coercions as a separate type

Progress

- Currently refactoring coercions as a separate type
- Fix newtype deriving bug!

Progress

- Currently refactoring coercions as a separate type
- Fix newtype deriving bug!
- Implement auto-lifting of non-GADTs

Outline

Allow lifting GADTs?

Closed type functions?

```
data Nat = Z | S Nat

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

Closed type classes?

```
class Foo (n::Nat) where
  ...

instance Foo Z where ...
instance Foo (S n) where ...
```

Proof search/induction?

```
Plus n Z ~ n
```

Lifting value-level functions to the type level?

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

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

Coming soon to a GHC near you!