System FC with Explicit Kind Equality

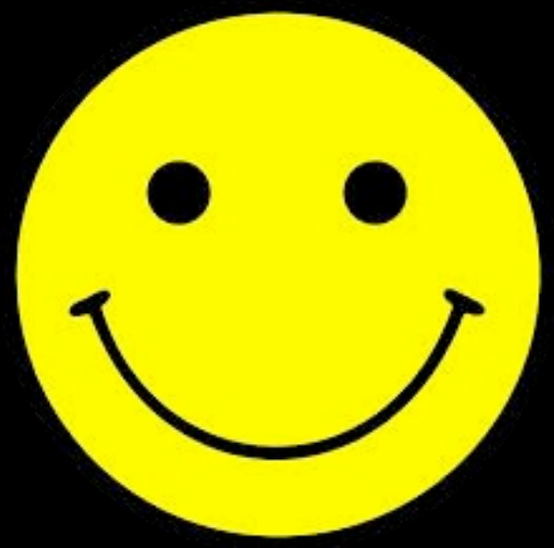
Stephanie Weirich

Justin Hsu

Richard A. Eisenberg

International Conference on Functional Programming
Thursday, September 26, 2013
Boston, MA, USA

Dependent types
+
Haskell



Disclaimer

No dependent types in Haskell, yet.

No dependent types in FC.

Yes: Support for dependently-typed programming using singletons in FC.

What we can do now

Generalized Algebraic Data Types (GADTs):

```
data Typ = TInt | TArrow Typ Typ

data Var :: [Typ] → Typ → ★ where
  VZero :: Var (a ‘: ctx) a
  VSucc :: Var ctx a → Var (b ‘: ctx) a

strengthen :: Var (b ‘: ctx) a
           → Maybe (Var ctx a)
strengthen VZero     = Nothing
strengthen (VSucc v) = Just v
```

Promotion in Haskell

```
data Typ = TInt | TArrow Typ Typ

data Var :: [Typ] → Typ → ★ where
  VZero :: Var (a ‘: ctx) a
  VSucc :: Var ctx a → Var (b ‘: ctx) a
```

```
ghci> :kind Var TInt TInt

The first argument of Var should have kind [Typ]
but TInt has kind Typ
```

Programming in types

Type-level functions:

```
type family Interpret (t :: Typ) :: ★
type instance Interpret TInt = Int
type instance Interpret (TArrow a b)
  = (Interpret a) → (Interpret b)
```

Kind polymorphism:

```
(':) :: ∀ k. k → [k] → [k]
```

WellScoped

```
data OutOfScope :: [Typ] → Nat → ★ where
  Oops :: OutOfScope ‘[] n
  Succ :: OutOfScope ctx n
       → OutOfScope (a ‘: ctx) (1 + n)
```

```
data WellScoped :: [Typ] → Nat → ★ where
  Yes :: ∀ (x :: Var ctx a).
         WellScoped ctx (EraseVar x)
  No  :: OutOfScope ctx n → WellScoped ctx n
```

```
Var of kind [Typ] → Typ → ★ is not promotable
```

Types vs. Kinds

Types	Kinds
Typ	Typ
(:) :: ∀a. a → [a] → [a]	(':) :: ∀k. k → [k] → [k]
Var	???
EraseVar	???

Need universal promotion of types to kinds

We need universal promotion to be able to express dependently-typed programs in Haskell.

How to proceed?

GHC compiles Haskell to System FC, a strongly-typed intermediate language

$\Rightarrow$

System FC must support universal promotion

System FC

- System FC must have decidable, fast type-checking
  - "System FC" = "System F with coercions"
  - ... but only type coercions
- Type coercions are used to...
  - ... implement GADTs
  - ... implement type families

GADTs to Coercions

Haskell

```
data Typ = TInt | TArrow Typ Typ
data Var :: [Typ] → Typ → ★ where
  VZero :: Var (a ‘: ctx) a
  VSucc :: Var ctx a → Var (b ‘: ctx) a
```

System FC

```
Typ    :: ★                    Typ    :: □
TInt   :: Typ                  TInt   :: Typ
TArrow :: Typ → Typ → Typ      TArrow :: Typ → Typ → Typ
Var    :: [Typ] → Typ → ★

VZero :: ∀ (ctx :: [Typ]) (a :: Typ). ∀ (ctx0 :: [Typ]).
         (ctx ~ (a ‘: ctx0)) → Var ctx a
VSucc :: ∀ (ctx :: [Typ]) (a :: Typ).
         ∀ (ctx0 :: [Typ]) (b0 :: Typ).
         (ctx ~ (b0 ‘: ctx0)) → Var ctx0 a → Var ctx a
```

GADT Pattern-match

Haskell

```
strengthen :: Var (b ‘: ctx) a
           → Maybe (Var ctx a)
strengthen VZero     = Nothing
strengthen (VSucc v) = Just v
```

System FC

```
VSucc :: ∀ (ctx :: [Typ]) (a :: Typ).
         ∀ (ctx0 :: [Typ]) (b0 :: Typ).
         (ctx ~ (b0 ‘: ctx0)) → Var ctx0 a → Var ctx a
```

In pattern match:

```
...
co :: (b ‘: ctx) ~ (b0 ‘: ctx0)
v :: Var ctx0 a
----------------------
Var ctx a
```

Answer:
Cast by a coercion built from co

If we want type-level GADTs, we need kind-level coercions.

Adding kind coercions is hard.

Merging types and kinds

$\tau ::=$		$\kappa ::=$	
α	variable	χ	variable
$\mid H$	constant	$\mid D \mid \star$	constants
$\mid \tau_1\ \tau_2$	application	$\mid \kappa_1\ \kappa_2$	application
$\mid \forall(\alpha{:}\kappa).\tau$	polymorphism	$\mid \forall\chi.\kappa$	polymorphism

$\tau, \kappa ::=$

$\mid \alpha$	variable
$\mid H \mid \star$	constants
$\mid \tau_1\ \tau_2$	application
$\mid \forall(\alpha{:}\kappa).\tau$	polymorphism
$\mid \ldots$	...

★ : ★

- What is ★'s type?
  - Common answer:
    infinite hierarchy of universes ($★_0$, $★_1$, $★_2$, ...)
  - Our answer: ★ : ★
- Isn't that dangerous?
  - Haskell is not a logic: all types are inhabited already
  - Type safety requires consistency of coercions
  - Proof of coercion consistency in paper

Heterogeneous Equality

- Consider:

  $\mathsf{id} :: \forall\,(a :: \star).\ a \to a$

  $\kappa :: \star$

  $\gamma_1 :: \mathsf{id} \sim \mathsf{id}$

  $\gamma_2 :: \kappa \sim \star$

  $\gamma_3 :: \mathsf{id}\ \kappa \sim \mathsf{id}\ \star$

  $\mathsf{id}\ \kappa :: \kappa \to \kappa$

  $\mathsf{id}\ \star :: \star \to \star$

- Thus, γ_3 is a heterogeneous coercion.
- Design option: do we allow these?
- Design decision: yes -- “John Major” equality

Our contributions

- Full details of enhanced System FC, supporting
  - universal promotion of datatypes
  - kind-level functions
  - kind-indexed GADTs (see paper)
- Operational semantics and "push rules"
  ⇒ lifting lemma, for the Preservation Theorem
- The consistency lemma: why Int ≠ Bool
  ⇒ necessary for the Progress Theorem
- Prototype implementation (Core language only)

Future work

II