Gadt: Almost Dependent Types

Florent Balestrieri

Department of Computer Science
University of Nottingham

FP Lab away day, 13 June 2013

Message

Do not underestimate the expressive power of Gadts.

The biggest limitation of Haskell and Omega that makes it painful to work with dependent type is the absence of type abstraction.

Motivation

Previous work

Some kind of limited type dependency is hacked into hackskell with

- Type classes
- Type families and gadts

My work

I show a systematic approach solely based on system F features, powerful enough to capture dependency on Π types.

```
cong : (a : Set) → (b : a → Set) → (f : Π a b)
→ ∀ x x′ → x = x′ → f x = f x′
```

GHC Extensions

- UnicodeSyntax
- TypeOperators
- ExistentialQuantification
- RankNTypes
- GADTs
- KindSignatures
- No type families!

Teaser

In Haskell, can we write something like that?

```
cong : (a : Set) → (b : a → Set) →
     → (f : Π a b) → (x : a) (x′ : a)
     → x = x′ → f x = f x′
```

What would the type in Haskell look like?

Teaser

In Haskell, can we write something like that?

```
cong : (a : Set) → (b : a → Set) →
    → (f : Π a b) → (x : a) (x′ : a)
    → x = x′ → f x = f x′
```

What would the type in Haskell look like?

```
cong :: Pi a b (R f) → a x → a x′
    → f x y → f x′ y′
    → Equal x x′ → Equal y y′
```

Teaser

In Haskell, can we write something like that?

```
cong : (a : Set) → (b : a → Set) →
     → (f : Π a b) → (x : a) (x′ : a)
     → x = x′ → f x = f x′
```

What would the type in Haskell look like?

```
cong :: ∀ (a :: ⋆) (b :: ⋆ → ⋆) .
     → Pi a b (R f) → a x → a x′
     → f x y → f x′ y′
     → Equal x x′ → Equal y y′
```

GADTs as relations

Unary relations representing sets

`Nat :: ⋆ → ⋆`

`p :: Nat n` is a proof that `n ∈ Nat`

Values like `p` are derivation trees.

Binary relations representing indexed sets

`Fin :: ⋆ → ⋆ → ⋆`

`p :: Fin n x` is a proof that `x ∈ Fin n`

Relations representing functions

`Plus :: ⋆ → ⋆ → ⋆ → ⋆`

`p :: Plus x y r` is a proof that $x + y = r$

Singleton Types

```
data N = Nz | Ns !N

data Nat n where
  Z :: Nat Z
  S :: Nat n → Nat (S n)
```

Bijections

```
up :: N → E Nat
down :: Nat n → N

up Nz       = E Z
up (Ns n)   = case up n of {E n′ → E (S n′)}
down Z      = Nz
down (S n)  = Ns (down n)
```

Dependent Inductive Types

Fin n x	iff			$x \in$ Fin n
List r x	iff			$x \in$ List r
Vec r n x	iff	$n \in$ Nat	&	$x \in$ Vec r n

Example

```
data Vec n r v where
  Vnil :: Vec r Z Nil
  Vcons :: r a → Vec r n v →
              Vec r (S n) (Cons a v)
```

Dependent Haskell Functions

```
nat_elim ::
  p Z →
  (∀ n . p n → p (S n)) →
  ∀ n . Nat n → p n
nat_elim pz ps Z       =  pz
nat_elim pz ps (S n)   =  ps (nat_elim pz ps n)
```

Functional Relations

Using

Represent computations with types

```
  Plus x y r    iff    x + y == r
  Length v n    iff  length v == n
Append u v w    iff    u ++ v == w
   Equal x y    iff     id x == y
```

Proving function properties

```
length (u ++ v) == length u + length v

appendPropLen :: Append u v w →
  Length u a →  Length v b → Length w c →
  Plus a b c′ → Equal c c′
```

Functional Relations

Proof that `x + y = r`

```
data Plus x y r where
  Pz :: Plus Z q q
  Ps :: Plus p q r → Plus (S p) q (S r)
```

Plus is a function

plusFun :: Plus a b r → Plus a b r' → Equal r r'

plus is total

plusTot :: Nat a → Nat b → Exist (Plus a b)

The range is `Nat`

plusNat :: Nat a → Nat b → Plus a b r → Nat r

Functional Relations

Defining

```
data Plus x y r where
  Pz :: Plus Z q q
  Ps :: Plus p q r → Plus (S p) q (S r)
```

Plus is a function

```
plusFun :: Plus a b r → Plus a b r′ → Equal r r′

plusFun Pz Pz = Refl
plusFun (Ps r) (Ps r′) = case plusFun r r′ of
  Refl → Refl
```

Functional Relations

Defining

```
data Plus x y r where
  Pz :: Plus Z q q
  Ps :: Plus p q r → Plus (S p) q (S r)
```

Plus is total

```
plusNatTot :: Nat a → Nat b → E (Plus a b)
plusNatTot Z       q = E $ Pz
plusNatTot (S n)   q = case plusNatTot n q of
  E p → E (Ps p)
```

Pi Types

Defining

Properties of functional relations

```
data IsPi a b f = IsPi
 { hasDomain   = ∀ x y. f x y → Sigma a b (x,y)
 , isFunction  = ∀ x y y'.
                   f x y → f x y' → Equal y y'
 , isTotal     = ∀ x. a x → E (f x)              }
```

A problem of kind

```
data R (f :: ⋆ → ⋆ → ⋆) where
  R :: R f
data Pi a b x where
  Pi :: IsPi a b f → Pi a b (R f)
```

Pi Types

Using

Properties of functional relations

```
data IsPi a b f = IsPi
 { hasDomain   = ∀ x y. f x y → Sigma a b (x,y)
 , isFunction = ∀ x y y′.
                f x y → f x y′ → Equal y y′
 , isTotal     = ∀ x. a x → E (f x)              }
```

Example

```
cong (Pi f) x x′ y y′ Refl = isFunction f y y′
cong :: Pi a b (R f) → a x → a x′
     → f x y → f x′ y′
     → Equal x x′ → Equal y y′
```

Type level representation of. . .

Sigma types

```
data Sigma a b t where
  Sigma :: a x → b x y → Sigma a b (x,y)
```

W Types

```
data W a b t where
  W :: a x → Fun (b x) (W a b) (R f)
     → W a b (x, (R f))
```

Perspective

Type classes

```
applyVal :: (HasRep x a, HasRep y b, TDF a b f)
         ⟹ Rel f → x → y
```

Applications

- Containers: W-functors and indexed W-functors in Haskell.
- Codes for strictly positive types, following Peter Morris' work: generic count, map, fold.

Conjecture

Can we embed λP (and $\lambda P2$) terms in $\lambda 2$, such that typing of each term in their respective system is equivalent?

Leibniz Equality

```
type a ≡ b = ∀ f . f a → f b
```

Leibniz Equality

```
type a ≡ b = ∀ f . f a → f b
```

Coercion

```
cast :: a ≡ b  →  (a → b)
cast eq = unId ∘ eq ∘ Id
newtype Id t = Id {unId :: t}
```

Leibniz Equality

```
type a ≡ b = ∀ f . f a → f b
```

Groupoid Operations

```
refl  :: a ≡ a
sym   :: a ≡ b → b ≡ a
trans :: a ≡ b → b ≡ c → a ≡ c

refl          = id
sym eq        = unSym $ eq $ Sym refl
trans ab bc = bc ∘ ab

newtype Sym a b = Sym {unSym :: b ≡ a}
```

Encoding Existentials

Algebraic datatype

```
data Exist t = ∀ a . E (t a)
```

Gadt

```
data Exist t where
  E :: t a → Exist t
```

Church encoding

```
type Exist′ t = ∀ r . (∀ x . t x → r) → r
```

Bijections

```
elimExist (E t) f = f t
exist with = with E

elimExist :: Exist t  → Exist′ t
exist     :: Exist′ t → Exist t
```

Encoding Existentials

Algebraic datatype

```
data Exist t = ∀ a . E (t a)
```

Gadt

```
data Exist t where
  E :: t a → Exist t
```

Church encoding

```
type Exist' t = ∀ r . (∀ x . t x → r) → r
```

Bijections

```
elimExist (E t) f = f t
exist with = with E

elimExist :: Exist t  → Exist' t
exist     :: Exist' t → Exist t
```

Encoding Gadts

Gadt

```
data Fin n where
  Fz :: Fin (S p)
  Fs :: Fin p → Fin (S p)
```

Algebraic datatype

```
data Fin′ n
 = ∀ p. Fz′ (n ≡ S p)
 | ∀ p. Fs′ (n ≡ S p) (Fin′ p)
```

Without existentials

```
data Fin″ n
 = Fz″ (∀ r.(∀ p. n ≡ S p → r) → r)
 | Fs″ (∀ r.(∀ p. n ≡ S p → Fin″ p → r) → r)
```

Equality type with a Gadt

```
data Equal a b where
  Refl :: Equal a a
```

Bijection with Leibniz equality

```
type a ≡ b = ∀ f . f a → f b

toEqual :: a ≡ b → Equal a b
toEqual e = e Refl

fromEqual :: Equal a b → a ≡ b
fromEqual Refl = refl
```

Advantages of gadts

Pattern matching a gadt will substitute and unify for you.

With Leibniz, we must carry equality proofs around to substitue.