Generalised Algebraic Data Types

Simon Peyton Jones
Microsoft Research
Geoffrey Washburn
University of Pennsylvania
Stephanie Weirich
University of Pennsylvania

A typical evaluator

```
data Term = Lit Int
          | Succ Term
          | IsZero Term
          | If Term Term Term

data Value = VInt Int | VBool Bool

eval :: Term -> Value
eval (Lit i) = VInt i
eval (Succ t) = case eval t of { VInt i -> VInt (i+1) }
eval (IsZero t) = case eval t of { VInt i -> VBool (i==0) }
eval (If b t1 t2) = case eval b of
                       VBool True -> eval t1
                       VBool False -> eval t2
```

Richer data types

What if you could define data types with richer return types?
Instead of this:

```
data Term where
    Lit :: Int -> Term
    Succ :: Term -> Term
    IsZero :: Term -> Term
    If :: Term -> Term -> Term -> Term
```

we want this:

```
data Term a where
    Lit :: Int -> Term Int
    Succ :: Term Int -> Term Int
    IsZero :: Term Int -> Term Bool
    If :: Term Bool -> Term a -> Term a -> Term a
```

Now (If (Lit 3) ...) is ill-typed.

Type evaluation

Now you can write a cool typed evaluator

```
eval :: Term a -> a
eval (Lit i)          = i
eval (Succ t)         = 1 + eval t
eval (IsZero i)       = eval i == 0
eval (If b e1 e2) = if eval b then eval e1 else eval e2
```

- You can't construct ill-typed terms
- Evaluator is easier to read and write
- Evaluator is more efficient too

What are GADTs?

Normal Haskell or ML data types:

```
data T a = T1 | T2 Bool | T3 a a
```

gives rise to constructors with types

```
T1 :: T a
T2 :: Bool -> T a
T3 :: a -> a -> T a
```

Return type is always (T a)

GADTs

Generalised Algebraic Data Types (GADTs):

- Single idea: allow arbitrary return type for constructors, provided outermost type constructor is still the type being defined

```
data Term a where
    Lit :: Int -> Term Int
    Succ :: Term Int -> Term Int
    IsZero :: Term Int -> Term Bool
    If :: Term Bool -> Term a -> Term a -> Term a
```

- Programmer gives types of constructors directly

GADTs have many names

- These things have been around a while, but are recently becoming popular in fp community
- Type theory (early 90's)
  - inductive families of datatypes
- Recent Language design
  - Guarded recursive datatypes (Xi et al.)
  - First-class phantom types (Hinze/Cheney)
  - Equality-qualified types (Sheard et al.)
  - Guarded algebraic datatypes (Simonet/Pottier)

GADts have many applications

- Language description and implementation

```
eval :: Term a -> a
step :: Config a -> Config a
```

  Subject reduction proof embedded in code for step!

- Domain-specific embedded languages

```
data Mag u where
  Pix :: Int -> Mag Pixel
  Cm :: Float -> Mag Centimetre

circle :: Mag u -> Region u
union :: Region u -> Region u -> Region u
tranform :: (Mag u -> Mag v) -> Region u -> Region v
```

More examples

- Generic programming

```
data Rep a where
        RInt :: Rep Int
        RList :: Rep a -> Rep [a]
        ...

zip :: Rep a -> a -> [Bit]
zip RInt i              = zipInt i
zip (RList r) []        = [0]
zip (RList r) (x:xs)    = 1 : zip r x ++ zip (RList r) xs
```

- Dependent types:

```
cons :: a -> List l a -> List (Succ l) a
head :: List (Succ l) a -> a
```

Just a modest extension?

Yes....

- Construction is simple: constructors are just ordinary polymorphic functions
- All the constructors are still declared in one place
- Pattern matching is still strictly based on the value of the constructor; the dynamic semantics can be type-erasing

Just a modest extension?

- But: Type checking
  Pattern matching
  is another matter

```
data Term a where
    Lit :: Int -> Term Int
    Succ :: Term Int -> Term Int
    IsZero :: Term Int -> Term Bool
    If :: Term Bool -> Term a -> Term a -> Term a
```

```
eval :: Term a -> a
eval (Lit i)          = i
eval (Succ t)         = 1 + eval t
eval (IsZero i)       = eval i == 0
eval (If b e1 e2) = if eval b then eval e1 else eval e2
```

In here, a=Int.
Notice rhs::Int

In here, a=Bool
Notice: rhs::Bool

- In a case alternative, we may know more about 'a'; we call this "type refinement"
- Result type is the anti-refinement of the type of each alternative

Our goal

- Add GADTs to Haskell
- Application of existing ideas -- but some new angles
- All existing Haskell programs still work
- Require some type annotations for pattern matching on GADTs
- But specify precisely what such annotations should be

Two steps

- Explicitly-typed System F-style language with GADTs
- Implicitly-typed source language (Simon's talk!)

Explicitly typed GADTs

Explicitly typed System F

Type abstraction and application

$$
\begin{array}{llcl}
\text{Variables} & x, y, z & & \\
\text{Constructors} & C & & \\
& & & \\
\text{Terms} & t, u & ::= & x \mid C_\sigma \\
& & \mid & \lambda x_\sigma \,.\, t \mid \Lambda\alpha \,.\, t \mid t\; u \mid t\; \sigma \\
& & \mid & \mathtt{let}\;\; x_\sigma \;=\; u \;\; \mathtt{in} \;\; t \\
& & \mid & \mathtt{case}(\sigma)\; t \;\mathtt{of}\; \overline{alt} \\
& & & \\
\text{Alternatives} & alt & ::= & p \rightarrow t \\
\text{Patterns} & p, q & ::= & C_\sigma\; \overline{\alpha}\; \overline{x_\sigma} \\
& & & \\
\text{Type variables} & a, b & & \\
\text{Type constructors} & T & & \\
\text{Types} & \sigma, \phi, \xi & ::= & \forall \alpha . \sigma \mid \sigma_1 \rightarrow \sigma_2 \\
& & \mid & T\; \overline{\sigma} \mid \alpha
\end{array}
$$

Explicitly typed binders

Result type of case

Patterns bind type variables

Impredicative

Patterns bind type variables

```
data Term a where
    Lit :: Int -> Term Int
    Succ :: Term Int -> Term Int
    IsZero :: Term Int -> Term Bool
    If :: Term Bool -> Term b -> Term b -> Term b
    Pair :: Term b -> Term c -> Term (b,c)
```

```
eval :: Term a -> a
eval a (x::Term a)
  = case(a) x of
       Lit (i::Int)              -> i
       Succ (t::Term Int)        -> 1 + eval Int t
       IsZero (i::Term Int)      -> eval Int i == 0

       Pair b c (t1::Term b) (t2::Term c) -> (eval b t1, eval c t2)

       If c (x::Term Bool) (e1::Term c) (e2::Term c)
            -> if eval Bool b then eval c e1 else eval c e2
```

Typing rules

Just exactly what you would expect....

$$\boxed{\Gamma \vdash t : \sigma}$$

$$\frac{}{\Gamma, x_\sigma \vdash x : \sigma}\ \text{VAR} \qquad \frac{}{\Gamma \vdash C_\sigma : \sigma}\ \text{CON} \qquad \frac{\Gamma \vdash t : \sigma' \to \sigma \qquad \Gamma \vdash u : \sigma'}{\Gamma \vdash t\ u : \sigma}\ \text{TERM-APP} \qquad \frac{\Gamma \vdash^k \sigma \qquad \Gamma \vdash t : \forall a\,.\,\sigma'}{\Gamma \vdash t\ \sigma : \sigma'[\sigma/a]}\ \text{TYPE-APP}$$

$$\frac{\Gamma \vdash^k \sigma \qquad \Gamma, x_\sigma \vdash t : \sigma'}{\Gamma \vdash (\lambda x_\sigma\,.\,t) : (\sigma \to \sigma')}\ \text{TERM-LAM} \qquad \frac{\Gamma, a \vdash t : \sigma}{\Gamma \vdash (\Lambda a\,.\,t) : \forall a\,.\,\sigma}\ \text{TYPE-LAM} \qquad \frac{\Gamma \vdash u : \sigma \qquad \Gamma, x_\sigma \vdash t : \sigma'}{\Gamma \vdash (\mathtt{let}\ \ x_\sigma{=}u\ \ \mathtt{in}\ \ t) : \sigma'}\ \text{LET}$$

...even for case expressions

$$\frac{\Gamma \vdash^{k} \sigma \qquad \Gamma \vdash t : \phi \qquad \Gamma \vdash^{\text{alt}} \overline{p \text{ -> } u} : \phi \rightarrow \sigma}{\Gamma \vdash (\text{case}(\sigma)\ t\ \text{of}\ \overline{p \text{ -> } u}) : \sigma}\ \text{CASE}$$

- Auxiliary judgement checks each alternative

Case alternatives

c is arity of C
t is arity of T

$$\boxed{\Gamma \vdash^{alt} p \text{ -> } t : \sigma_1 \rightarrow \sigma_2}$$

Instantiate with fresh type variables

$$\frac{\begin{array}{cc}(C : \forall \overline{\alpha}.\overline{\sigma}^{c} \rightarrow T\,\overline{\xi}^{t}) \in \Gamma & \overline{\alpha} \# \mathrm{dom}(\Gamma) \\ \theta \text{ is a partial unifier of } T\,\overline{\xi'}^{t} \text{ and } T\,\overline{\xi}^{t} & \theta(\Gamma, \overline{\alpha}, \overline{x : \sigma}^{c}) \vdash \theta(u) : \theta(\sigma)\end{array}}{\Gamma \vdash^{alt} C\,\overline{\alpha}\,\overline{x_{\sigma}}^{c} \rightarrow u : T\,\overline{\xi'}^{t} \rightarrow \sigma}\ \text{ALT-CON}$$

Unify constructor result type with context type

Apply unifier to this alternative

Observations:

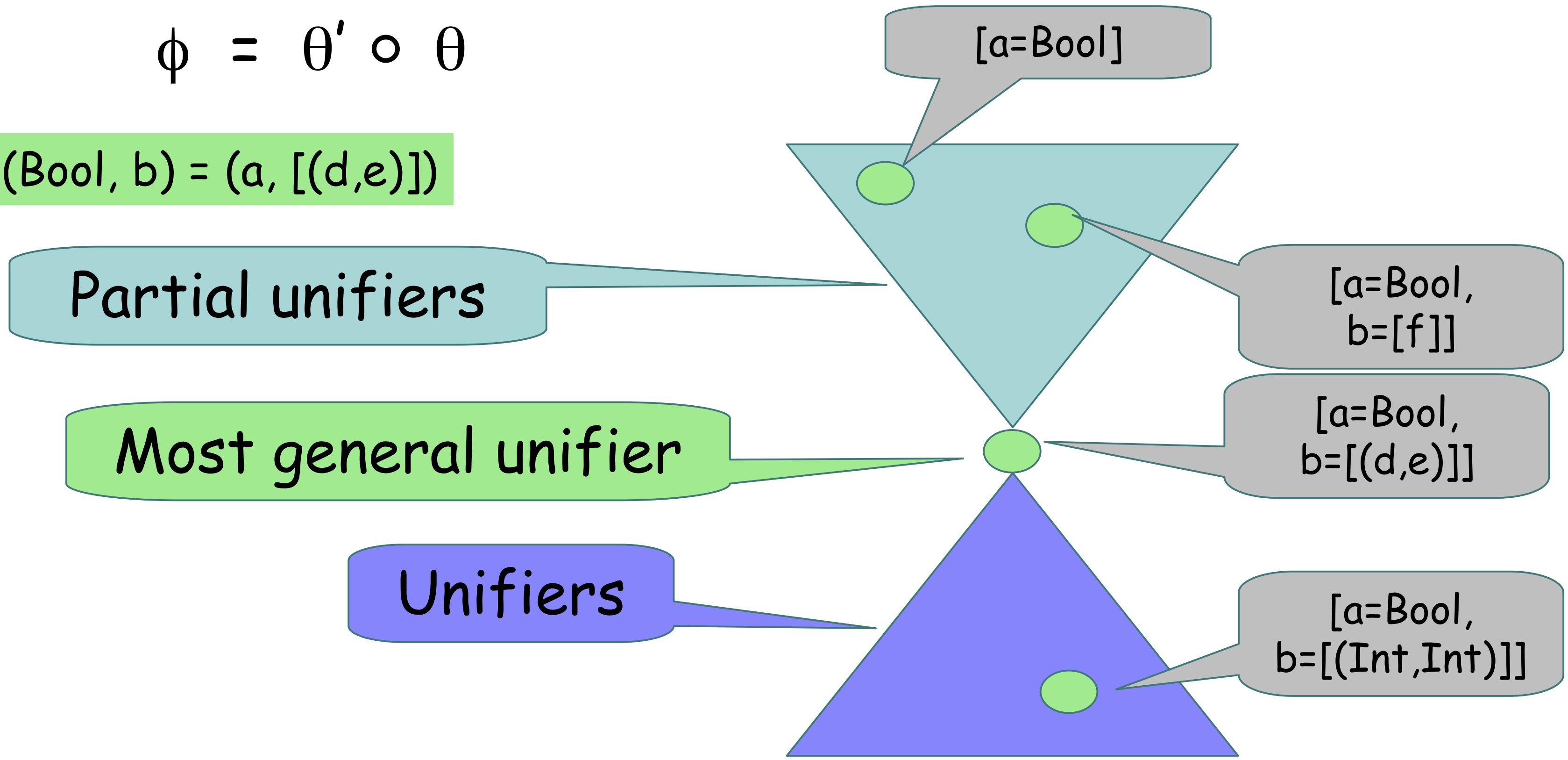
- Constructing unifier and applying it is equivalent to typing RHS in the presence of the refining constraint
- Unification works fine over polymorphic types

Partial unifiers

Definition. θ is a **partial unifier** of σ_1 and σ_2 iff for any unifier ϕ of σ_1 and σ_2 there is a substitution θ' such that

$$\phi \;=\; \theta' \circ \theta$$

E.g. (Bool, b) = (a, [(d,e)])



Case alternatives

$$\frac{(C : \forall\overline{\alpha}.\overline{\sigma}^c \to T\,\overline{\xi}^t) \in \Gamma \qquad \overline{\alpha}\,\#\,\mathrm{dom}(\Gamma) \qquad \theta \text{ is a partial unifier of } T\,\overline{\xi'}^t \text{ and } T\,\overline{\xi}^t \qquad \theta(\Gamma, \overline{\alpha}, \overline{x : \sigma}^c) \vdash \theta(u) : \theta(\sigma)}{\Gamma \vdash^{alt} C\,\overline{\alpha}\,\overline{x_\sigma}^c \to u : T\,\overline{\xi'}^t \to \sigma}\ \text{ALT-CON}$$

```
eval :: Term a -> a
eval a (x::Term a)
  = case(a) x of
      Lit (i::Int)            -> i
      Succ (t::Term Int)      -> 1 + eval Int t
      IsZero (i::Term Int)    -> eval Int i == 0

      Pair b c (t1::Term b) (t2::Term c) -> (eval b t1, eval c t2)

      If c (x::Term Bool) (e1::Term c) (e2::Term c)
                              -> if eval Bool b then eval c e1 else eval c e2
```

{a->Int}

{a->Bool}

{a->(b,c)}

{a->c} or {c->a}

A heffalump trap

$$\frac{\begin{array}{cc}(C : \forall\overline{\alpha}.\overline{\sigma}^{c} \to T\,\overline{\xi}^{t}) \in \Gamma & \overline{\alpha}\,\#\,dom(\Gamma) \\ \theta \text{ is a partial unifier of } \tau \text{ and } T\,\overline{\xi}^{t} & \theta(\Gamma, \overline{\alpha}, \overline{x : \sigma}^{c}) \vdash \theta(u) : \theta(\sigma)\end{array}}{\Gamma \vdash^{alt} C\,\overline{\alpha}\,\overline{x_{\sigma}}^{c} \to u : \tau \to \sigma}\ \text{ALT-CON}$$

```
\(x:a). case x of
          True -> False
          False -> True
```

- This should jolly well be rejected! (Or: forget Haskell and treat all constructors as drawn from some universal data type.)
- Conclusion: the outermost type constructor is special

Case alternatives

$$\boxed{\Gamma \vdash^{alt} p \to t : \sigma_1 \to \sigma_2}$$

$$\frac{\begin{array}{cc}(C : \forall \overline{\alpha}.\overline{\sigma^c} \to T\,\overline{\xi}^t) \in \Gamma & \overline{\alpha} \# dom(\Gamma) \\ \theta \text{ is a partial unifier of } T\,\overline{\xi'}^t \text{ and } T\,\overline{\xi}^t & \theta(\Gamma, \overline{\alpha}, \overline{x : \sigma^c}) \vdash \theta(u) : \theta(\sigma)\end{array}}{\Gamma \vdash^{alt} C\,\overline{\alpha}\,\overline{x_\sigma}^c \to u : T\,\overline{\xi'}^t \to \sigma}\ \text{ALT-CON}$$

$$\frac{(C : \forall \overline{\alpha}.\overline{\sigma^c} \to T\,\overline{\xi}^t) \in \Gamma \qquad \overline{\alpha} \# dom(\Gamma) \qquad T\,\overline{\xi'}^t \text{ and } T\,\overline{\xi}^t \text{ have no unifier}}{\Gamma \vdash^{alt} C\,\overline{\alpha}\,\overline{x_\sigma}^c \to u : T\,\overline{\xi'}^t \to \sigma}\ \text{ALT-FAIL}$$

If unification fails,
ignore RHS
altogether

- Failure case needed for subject reduction

Nested patterns

Nested patterns

$$
\begin{array}{llll}
\text{Alternatives} & \mathit{alt} & ::= & p \to t \\
\text{Patterns} & p, q & ::= & x_\sigma \mid C_\sigma\, \overline{a}\, \overline{p} \\
\text{Constraint} & \pi & ::= & \sigma_1 \doteq \sigma_2 \\
\text{Constraint lists} & \Pi & ::= & \epsilon \mid \pi, \Pi
\end{array}
$$

$$
\frac{\Gamma;\epsilon;\emptyset \vdash^{p} p : \sigma_1 ; \Delta ; \theta \qquad \theta(\Gamma,\Delta) \vdash \theta(u) : \theta(\sigma_2)}{\Gamma \vdash^{a} p \to u : \sigma_1 \to \sigma_2}\ \text{ALT}
$$

$$
\boxed{\text{Patterns}\ \ \Gamma;\Delta;\theta \vdash^{p} p : \sigma ; \Delta' ; \theta'}
$$

Extend substitution θ
and bindings Δ

Patterns $\Gamma;\Delta;\theta \vdash^{p} p : \sigma;\Delta';\theta'$

$$\frac{x \,\#\, dom(\Delta) \qquad \theta(\sigma) = \theta(\phi)}{\Gamma;\Delta;\theta \vdash^{p} x_{\sigma} : \phi;\Delta,(x:\sigma);\theta}\ \text{PVAR}$$

$$\frac{\begin{array}{c}(C : \forall\overline{\alpha}.\overline{\sigma}^{c} \to T\,\overline{\xi}^{t}) \in \Gamma \qquad \overline{\alpha} \,\#\, dom(\Gamma,\Delta) \\ \theta(\phi) = T\,\overline{\xi'}^{t} \qquad \theta' = \mathcal{U}(T\,\overline{\xi}^{t} \doteq T\,\overline{\xi'}^{t}) \\ \Gamma;(\Delta,\overline{\alpha});\theta'\circ\theta \vdash^{ps} \overline{p:\sigma}^{c};\Delta'';\theta''\end{array}}{\Gamma;\Delta;\theta \vdash^{p} C\,\overline{\alpha}\,\overline{p}^{c} : \phi;\,\Delta'';\theta''}\ \text{PCON}$$

Avoid heffalump trap

Sadly, we cannot require ϕ to be of form $T\,\xi$, as we did before

Thread substitution through sub-patterns

Nested patterns

```
data Term a where
    Lit :: Int -> Term Int
    Succ :: Term Int -> Term Int
    IsZero :: Term Int -> Term Bool
```

$$\frac{\Gamma;\epsilon;\emptyset \vdash^{p} p : \sigma_1 ; \Delta ; \theta \qquad \theta(\Gamma,\Delta) \vdash \theta(u) : \theta(\sigma_2)}{\Gamma \vdash^{a} p \rightarrow u : \sigma_1 \rightarrow \sigma_2}\ \text{ALT}$$

Three possible outcomes:

- Success, producing substitution.
- Failure ($\theta=\perp$): this alternative cannot match
  e.g. \(x::Term Int) -> case x of { IsZero a -> a; ... }
- Type error: the program is rejected
  e.g. case 4 of { True -> 0; ... }

The source language

The ground rules

- Programmer-supplied type annotations are OK
- Whether or not a program is typeable will depend on type annotations
- The language specification should nail down exactly what type annotations are sufficient (so that if Compiler A accepts the program, then so will Compiler B)
- The language specification should not be a type inference algorithm

Polymorphic recursion

```
data Tree a = MkTree a (Tree (Tree a))

collect :: Tree a -> [a]
collect (MkTree x t) = x : concatMap collect (collect t)

concatMap :: (a->[b]) -> [a] -> [b]
```

Polymorphic recursion

```
data Tree a = MkTree a (Tree (Tree a))

collect :: Tree a -> [a]
collect a (MkTree x t) = x : concatMap (collect a) (collect (Tree a) t)
                                                    :: [Tree a]
concatMap :: (a->[b]) -> [a] -> [b]
```

- Hard to infer types from un-annotated program
- Dead easy to do so with annotation
- Express by giving two type rules for letrec f=e:
  - one for un-decorated decl: extend envt with (f::τ)
  - one for annotated decl: extend envt with (f:σ)

Goal

- The typing rules should exclude too-lightly-annotated programs, so that the remaining programs are "easy" to infer
- Type annotations should propagate, at least in "simple" ways

```
eval :: Term a -> a
eval (Lit i)          = i
eval (Succ t)         = 1 + eval t
eval (IsZero i)       = eval i == 0
eval (If b e1 e2) = if eval b then eval e1 else eval e2
```

Here information propagates from the type signature into the pattern and result types

Syntax

Type annotations on terms

Atoms	v	::=	$x \mid C$
Terms	t, u	::=	$v \mid \lambda p.t \mid t\ u \mid t::ty$
		|	$\mathtt{let}\ x = u\ \mathtt{in}\ t$
		|	$\mathtt{letrec}\ x::ty = u\ \mathtt{in}\ t$
		|	$\mathtt{case}\ t\ \mathtt{of}\ \overline{p \to t}$
Patterns	p, q	::=	$x \mid C\ \overline{p}$
Source types	ty	::=	$a \mid ty_1 \text{->} ty_2 \mid T\ \overline{ty}$
		|	$\mathtt{forall}\ \overline{a}.\ ty$
Polytypes	σ, ϕ	::=	$\forall \overline{\alpha}.\tau$
Monotypes	τ, υ	::=	$T\overline{\tau} \mid \tau_1 \to \tau_2 \mid \alpha \mid \boxed{\tau}$

No compulsory types on binders, or on case

Source types are part of syntax of programs

Internal types are stratified into **polytypes** and **monotypes**. All predicative

Syntax

$$
\begin{array}{lrcl}
\text{Atoms} & \nu & ::= & x \mid C \\
\text{Terms} & t, u & ::= & \nu \mid \lambda p . t \mid t\ u \mid t :: ty \\
 & & \mid & \texttt{let}\ x = u\ \texttt{in}\ t \\
 & & \mid & \texttt{letrec}\ x :: ty = u\ \texttt{in}\ t \\
 & & \mid & \texttt{case}\ t\ \texttt{of}\ \overline{p\ \texttt{->}\ t} \\
\text{Patterns} & p, q & ::= & x \mid C\ \overline{p} \\
\text{Source types} & ty & ::= & a \mid ty_1 \texttt{->} ty_2 \mid T\ \overline{ty} \\
 & & \mid & \texttt{forall}\ \overline{a}.\ ty
\end{array}
$$

$$
\begin{array}{lrcl}
\text{Polytypes} & \sigma, \phi & ::= & \forall \overline{\alpha} . \tau \\
\text{Monotypes} & \tau, \upsilon & ::= & T\overline{\tau} \mid \tau_1 \to \tau_2 \mid \alpha \mid \boxed{\tau}
\end{array}
$$

Exciting new feature:
wobbly types

IDEA 1: Wobbly types

- Simple approach to type-check case expressions:
  - form MGU as specified in rule
  - apply to the environment and RHS
  - type-check RHS
- Problem: in type inference, the types develop gradually, by unification

```
\x. (foo x, case x of
                Succ t    -> 1
                IsZero i -> 1 + True)
```

```
foo :: Term Int -> Bool
```

- Type inference guesses (x:a56), then (foo x) forces a56=Term Int, so the IsZero case can't match

Wobbly types

```
\x. (foo x, case x of
          Succ t    -> 1
          IsZero i -> 1 + True)
```

- We do not want the order in which the type inference algorithm traverses the tree to affect what programs are typeable.
- MAIN IDEA: boxes indicate guess points

$$\frac{\Gamma, (x : \boxed{\tau_1}) \vdash t : \tau_2}{\Gamma \vdash (\backslash x . t) : (\boxed{\tau_1} \to \tau_2)}$$

Box indicates a prescient guess by the type system

Wobbly types: intuition

- **Wobbly types correspond precisely to the places where a type inference algorithm allocates a fresh meta variable**
- The type system models only the place in the type where the guess is made, not the way in which it is refined by unification

Effect of wobbly types

- Wobbly types do not affect "normal Damas-Milner" type inference

- Wobbly types do not contribute to a type refining substitution:

$$\boxed{\text{Unification} \quad \vdash^{u} \Pi \leadsto \theta}$$

Effect of wobbly types

- Wobbly types are impervious to a type-refining substitution

$$
\begin{array}{rcll}
\theta(\tau) & :: & \tau & \text{Apply a type refinement} \\
\theta(\alpha) & = & \alpha & \text{if } \alpha \notin dom(\theta) \\
 & = & \tau & \text{if } [\alpha \mapsto \tau] \in \theta \\
\theta(T\,\overline{\tau}) & = & T\,\overline{\theta(\tau)} & \\
\theta(\tau_1 \rightarrow \tau_2) & = & \theta(\tau_1) \rightarrow \theta(\tau_2) & \\
\theta(\boxed{\tau}) & = & \tau &
\end{array}
$$

```
\(x::Term a). \y. case x of { ... }
```

y will get a boxed type, which will not be refined

IDEA 2: directionality flag δ

```
eval :: Term a -> a
eval = \x. case x of
              Lit i    -> i
              Succ t -> 1 + eval t
              ...etc...
```

- We want the type annotation on eval to propagate to the \x.

Directionality flags

Local Type Inference (Pierce/Turner)

$\Gamma \vdash_{\Uparrow} t : \tau$ In environment Γ, term t has type τ

$\Gamma \vdash_{\Downarrow} t : \tau$ In environment Γ and supplied context τ, term t is well-typed

$$\frac{\Gamma, (x : \boxed{\tau_1}) \vdash_{\Uparrow} t : \tau_2}{\Gamma \vdash_{\Uparrow} (\backslash x . t) : (\boxed{\tau_1} \to \tau_2)}$$

Guess

$$\frac{\Gamma, (x : \tau_1) \vdash_{\Downarrow} t : \tau_2}{\Gamma \vdash_{\Downarrow} (\backslash x . t) : (\tau_1 \to \tau_2)}$$

No guess

Typechecking functions

$$\frac{\Gamma \vdash_{\Uparrow} t : \tau \qquad \tau = \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash_{\Downarrow} u : \tau_1}{\Gamma \vdash_{\delta} t\, u : \tau_2}$$

Guess the function type (probably from Γ)

Check the argument type

So if f :: Term Int -> Int
then in the call (f e), we use checking mode for e

Higher rank types

- Directionality flags are used in a very similar way to propagate type annotations for higher rank types.
- Happy days! Re-use of existing technology!
- Shameless plug: “Practical type inference for arbitrary rank types”, on my home page
    http://research.microsoft.com/~simonpj

Bore 1: must "look through" wobbles

$$\frac{\Gamma \vdash_{\Uparrow} t : \tau \qquad \boxed{\mathrm{push}(\tau) = \tau_1 \to \tau_2} \qquad \Gamma \vdash_{\Downarrow} u : \tau_1}{\Gamma \vdash_{\delta} t\, u : \tau_2}$$

$\boldsymbol{\tau}$ might not be an arrow type: it might wobbly!

$$\begin{array}{rcl}
\mathrm{push}(\tau) & :: & \tau \\
\mathrm{push}(\boxed{T\ \overline{\tau}}) & = & T\ \overline{\boxed{\tau}} \\
\mathrm{push}(\boxed{\tau_1 \to \tau_2}) & = & \boxed{\tau_1} \to \boxed{\tau_2} \\
\mathrm{push}(\boxed{\boxed{\tau}}) & = & \mathrm{push}(\boxed{\tau})
\end{array}$$

Bore 2: guess meets check

$$\frac{\Gamma \vdash_{\Uparrow} t : \tau \qquad push(\tau) = \tau_1 \to \tau_2 \qquad \Gamma \vdash_{\Downarrow} u : \tau_1 \qquad \boxed{\vdash^{inst\tau}_{\delta} \tau_2 \sim \tau'_2}}{\Gamma \vdash_{\delta} t\, u : \tau'_2}$$

- Guessing mode is easy: $\tau_2 = \tau'_2$
- Checking mode is trickier: τ_2 might have different boxes than τ'_2

  We want $strip(\tau_2) = strip(\tau'_2)$

Bore 2: guess meets check

$$\frac{\Gamma \vdash_{\Uparrow} t : \tau \qquad \mathrm{push}(\tau) = \tau_1 \to \tau_2 \qquad \Gamma \vdash_{\Downarrow} u : \tau_1 \qquad \boxed{\vdash^{\mathrm{inst}\tau}_{\delta} \tau_2 \sim \tau_2'}}{\Gamma \vdash_{\delta} t\, u : \tau_2'}$$

$$\boxed{\vdash^{\mathrm{inst}\tau}_{\delta} \tau \sim \tau}$$

$$\frac{}{\vdash^{\mathrm{inst}\tau}_{\Uparrow} \tau \sim \tau}\ \mathrm{INST}\tau\Uparrow \qquad \frac{\mathrm{strip}(\tau) = \mathrm{strip}(\upsilon)}{\vdash^{\mathrm{inst}\tau}_{\Downarrow} \tau \sim \upsilon}\ \mathrm{INST}\tau\Downarrow$$

$$\begin{array}{rcl}
\mathrm{strip}(\alpha) & = & \alpha \\
\mathrm{strip}(T\,\overline{\tau}) & = & T\,\overline{\mathrm{strip}(\tau)} \\
\mathrm{strip}(\tau_1 \to \tau_2) & = & \mathrm{strip}(\tau_1) \to \mathrm{strip}(\tau_2) \\
\mathrm{strip}(\boxed{\tau}) & = & \mathrm{strip}(\tau)
\end{array}$$

The good news

$$\frac{\Gamma \vdash_{\Uparrow} u : \tau_1 \qquad \Gamma \vdash^{a}_{\delta} \overline{\mathtt{p\text{->}t} : \tau_1 \rightarrow \tau_2}}{\Gamma \vdash_{\delta} (\mathtt{case}\ u\ \mathtt{of}\ \overline{\mathtt{p\text{->}t}}) : \tau_2}\ \text{CASE}$$

- Just like before, modulo passing on directionality flags

Abstraction

- Lambdas use the same auxiliary judgement as case

Guess here

$$\frac{\Gamma \vdash^{k} \tau_1 \qquad \Gamma \vdash^{a}_{\Uparrow} \texttt{p->t} : \boxed{\tau_1} \to \tau_2}{\Gamma \vdash_{\Uparrow} \lambda \texttt{p}.\texttt{t} : \boxed{\tau_1} \to \tau_2} \text{ ABS}\Uparrow$$

$$\frac{\Gamma \vdash^{a}_{\Downarrow} \texttt{p->t} : \tau_1 \to \tau_2}{\Gamma \vdash_{\Downarrow} \lambda \texttt{p}.\texttt{t} : \tau_1 \to \tau_2} \text{ ABS}\Downarrow$$

Case alternatives

$$\boxed{\text{Case alternatives } \Gamma \vdash_{\delta}^{a} p \text{->} u : \tau_1 \rightarrow \tau_2}$$

$$\frac{\Gamma;\epsilon;\emptyset \vdash^{p} p : \tau_1;\ \Delta;\ \theta \qquad \mathrm{dom}(\Delta) \,\#\, (\mathit{ftv}(\tau_2)) \qquad \theta(\Gamma,\Delta) \vdash_{\delta} u : \boxed{\theta_{\delta}(\tau_2)}}{\Gamma \vdash_{\delta}^{a} p \text{->} u : \tau_1 \rightarrow \tau_2}\ \text{ALT}$$

Only refine result type when in checking mode

$$\begin{aligned} \theta_{\Uparrow}(\tau) &= \tau \\ \theta_{\Downarrow}(\tau) &= \theta(\tau) \end{aligned}$$

Patterns

$$\boxed{\text{Patterns } \Gamma;\Delta;\theta \vdash^{p} p : \tau;\Delta';\theta'}$$

Bindings and type refinement from "earlier" patters

Augmented with bindings and type refinements from p

Patterns

$$\frac{\begin{array}{c}(\mathrm{C} : \forall\overline{\alpha}.\overline{\tau}^{c} \to \mathrm{T}\,\overline{\upsilon}^{t}) \in \Gamma \qquad \overline{\alpha} \,\#\, \mathrm{dom}(\Gamma,\Delta) \\ \boxed{\mathrm{push}(\theta(\upsilon')) = \mathrm{T}\,\overline{\upsilon''}^{t}} \quad \boxed{\vdash^{u} (\mathrm{T}\,\overline{\upsilon}^{t} \doteq \mathrm{T}\,\overline{\upsilon''}^{t}) \rightsquigarrow \theta'} \\ \Gamma;\Delta,\overline{\alpha};(\theta' \circ \theta) \vdash^{ps} \overline{p : \tau}^{c} ; \Delta'';\theta''\end{array}}{\Gamma;\Delta;\theta \vdash^{p} \mathrm{C}\,\overline{p}^{c} : \upsilon';\Delta'';\theta''}\ \text{PCON}$$

Ensure the pattern type has the right shape

Same as before except...

Perform wobbly unification

Wobbly unification

$$\boxed{\text{Unification} \quad \vdash^{u} \Pi \rightsquigarrow \theta}$$

- Goal: θ makes the best refinement it can using only the **rigid** parts of Π
- A type is "rigid" if it has no wobbly parts.

$$\frac{\theta(\Pi') = \Pi \qquad \mathrm{dom}(\theta)\,\#\,\mathrm{ftv}(\Pi) \qquad \Pi' \text{ is rigid} \qquad \theta' \text{ is a most general unifier of } \Pi'}{\vdash^{u} \Pi \rightsquigarrow (\theta \circ \theta')\,|_{\mathrm{ftv}(\Pi)}}\ \text{UNIF}$$

Soundness of the source

- The type system is sound
- Proved by type-directed translation in the core language

$$\text{THEOREM 4.1. } \textit{If } \Gamma \vdash_\delta \mathrm{t} \rightsquigarrow \mathrm{t}' : \tau \textit{ then } \mathcal{S}(\Gamma) \vdash \mathrm{t}' : \mathcal{S}(\tau)$$

Our typing judgements also do a type-directed translation

Strip boxes

Core-language judgement

Conclusions

- Wobbly types seem new
- Rigid types mean there is a programmer-explicable "audit trail" back to a programmer-supplied annotation
- Resulting type system is somewhat complicated, but much better than "add annotations until the compiler accepts the program"
- Claim: does "what the programmer expects"
- Implementing in GHC now

http://research.microsoft.com/~simonpj

MGU

$$\boxed{\Gamma \vdash^{alt} p \text{ -> } t : \phi \rightarrow \sigma}$$

$$\frac{\phi = \forall \overline{a}\,.\,\overline{\sigma}^{c} \rightarrow T\,\overline{\sigma'}^{t} \qquad \overline{a} \notin \Gamma \qquad \theta = MGU(T\,\overline{\xi}^{t} \doteq T\,\overline{\sigma'}^{t}) \qquad \theta(\Gamma, \overline{a}, \overline{x_{\sigma}}^{c}) \vdash \theta(u) : \theta(\sigma)}{\Gamma \vdash^{alt} C_{\phi}\;\overline{a}\;\overline{x_{\sigma}}^{c} \rightarrow u : T\,\overline{\xi}^{t} \rightarrow \sigma} \text{ ALT-CON}$$

- Must θ be the most-general unifier in a sound typing rule?
- Yes and no: It does not have to be a unifier, but it must be "most general".
- θ is a partial unifier of Π iff

  for any unifier Φ of Π, there is a substitution θ' such that: Φ = θ' o θ