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 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:

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

we want this:

```haskell
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

```haskell
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:

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

gives rise to constructors with types

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

Return type is always \((T a)\)
Generalised Algebraic Data Types (GADTs):

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

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

```haskell
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
```

```haskell
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 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

Variables: $x, y, z$

Constructors: $C$

Terms: $t, u ::= x | C_\sigma | \lambda x_\sigma . t | \Lambda \alpha . t | tu | t_\sigma$

Explicitly typed binders:

Alternatives: $alt ::= p \rightarrow t$

Patterns: $p, q ::= C_\sigma \overline{\alpha} \overline{x_\sigma}$

Type variables: $a, b$

Type constructors: $\top$

Types: $\sigma, \phi, \xi ::= \forall \alpha . \sigma | \sigma_1 \rightarrow \sigma_2 | \top \overline{\sigma} | \alpha$

Impredicative

Type abstraction and application

Result type of case

Patterns bind type variables
Patterns bind type variables

```haskell
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....
...even for case expressions

\[
\frac{\Gamma \vdash^k \sigma \quad \Gamma \vdash t : \phi \quad \Gamma \vdash^{alt} \overline{p \rightarrow u} : \phi \rightarrow \sigma}{\Gamma \vdash \text{case}(\sigma) \ t \ \text{of} \ \overline{p \rightarrow u} : \sigma}
\]

- Auxiliary judgement checks each alternative
Case alternatives

\[
\Gamma \vdash_{\text{alt}} p \rightarrow t : \sigma_1 \rightarrow \sigma_2
\]

\[
(C : \forall \alpha. \sigma^c \rightarrow T \bar{\xi}^t) \in \Gamma \quad \bar{\alpha} \# \text{dom}(\Gamma)
\]

\[
\theta \text{ is a partial unifier of } T \bar{\xi'}^t \text{ and } T \bar{\xi}^t \quad \theta(\Gamma, \bar{\alpha}, x : \sigma^c) \vdash \theta(u) : \theta(\sigma)
\]

\[
\Gamma \vdash_{\text{alt}} C \bar{\alpha} \bar{x} \sigma^c \rightarrow u : T \bar{\xi'}^t \rightarrow \sigma
\]

Observations:

- Constructing unifier and applying it is equivalent to typing RHS in the presence of the refining constraint
- Unification works fine over polymorphic types
Definition. \( \theta \) is a **partial unifier** of \( \sigma_1 \) and \( \sigma_2 \) iff for any unifier \( \phi \) of \( \sigma_1 \) and \( \sigma_2 \) there is a substitution \( \theta' \) such that \( \phi = \theta' \circ \theta \)

E.g. \((\text{Bool}, b) = (a, [(d,e)])\)
Case alternatives

\[(C : \forall \alpha. \sigma^c \rightarrow T \bar{\xi}^t) \in \Gamma \quad \alpha \not\in \text{dom}(\Gamma) \]
\[
\theta \text{ is a partial unifier of } T \bar{\xi}_i^t \text{ and } T \bar{\xi}_j^t \quad \theta(\Gamma, \bar{\alpha}, \bar{x} : \sigma^c) \vdash \theta(u) : \theta(\sigma) \]
\[
\Gamma \vdash_{\text{alt}} C \bar{\alpha} \bar{x}_\sigma^c \rightarrow u : T \bar{\xi}_i^t \rightarrow \sigma
\]

\[
\text{eval} :: \text{Term } a \rightarrow a
\]
\[
\text{eval } a \ (x::\text{Term } a)
\]
\[
= \text{case}(a) \times \text{of}
\]
\[
\text{Lit } (i::\text{Int}) \rightarrow i
\]
\[
\text{Succ } (t::\text{Term } \text{Int}) \rightarrow 1 + \text{eval } \text{Int } t
\]
\[
\text{IsZero } (i::\text{Term } \text{Int}) \rightarrow \text{eval } \text{Int } i == 0
\]
\[
\text{Pair } b \ c \ (t1::\text{Term } b) \ (t2::\text{Term } c) \rightarrow \text{eval } b \ t1, \text{eval } c \ t2
\]
\[
\text{If } c \ (x::\text{Term } \text{Bool}) \ (e1::\text{Term } c) \ (e2::\text{Term } c)
\]
\[
\rightarrow \text{if eval } \text{Bool } b \text{ then eval } c \ e1 \text{ else eval } c \ e2
\]
A heffalump trap

\[(C : \forall \bar{\alpha}. \sigma^c \rightarrow T \bar{\xi}^t) \in \Gamma \quad \bar{\alpha} \not\in \text{dom}(\Gamma)\]
\[
\theta \text{ is a partial unifier of } \tau \text{ and } T \bar{\xi}^t
\]
\[
\Gamma \vdash_{\text{alt}} C \bar{\alpha} \bar{x}_{\sigma^c} \rightarrow u : \tau \rightarrow \sigma
\]

\((x:a). \text{ case } x \text{ of }
\begin{align*}
\text{True} & \rightarrow \text{False} \\
\text{False} & \rightarrow \text{True}
\end{align*}\)

- 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

\[
\Gamma \vdash^{\text{alt}} p \rightarrow t : \sigma_1 \rightarrow \sigma_2
\]

- Failure case needed for subject reduction

\[
\begin{align*}
(C : \forall \alpha. \sigma^c \rightarrow T \xi^t) & \in \Gamma & \bar{\alpha} \not\in \text{dom}(\Gamma) \\
\theta & \text{ is a partial unifier of } T \xi'^t \text{ and } T \xi^t & \theta(\Gamma, \bar{\alpha}, x : \sigma^c) \vdash \theta(u) : \theta(\sigma) \\
\Gamma \vdash^{\text{alt}} C \bar{\alpha} x_\sigma^c \rightarrow u : T \xi'^t \rightarrow \sigma & \text{ ALT-CON}
\end{align*}
\]

\[
\begin{align*}
(C : \forall \alpha. \sigma^c \rightarrow T \xi^t) & \in \Gamma & \bar{\alpha} \not\in \text{dom}(\Gamma) & \text{ ALT-FAIL}
\end{align*}
\]

If unification fails, ignore RHS altogether
Nested patterns
Nested patterns

Alternatives

\[ \text{alt} ::= p \rightarrow t \]

Patterns

\[ p, q ::= \chi_\sigma | C_\sigma \overline{a} \overline{p} \]

Constraint

\[ \pi ::= \sigma_1 \Downarrow \sigma_2 \]

Constraint lists

\[ \Pi ::= \epsilon | \pi, \Pi \]

\[ \Gamma; \epsilon; \emptyset \vdash^p p : \sigma_1; \Delta; \theta \quad \theta(\Gamma, \Delta) \vdash \theta(u) : \theta(\sigma_2) \]

\[ \Gamma \vdash^a p \rightarrow u : \sigma_1 \rightarrow \sigma_2 \]

Patterns

\[ \Gamma; \Delta; \theta \vdash^p p : \sigma; \Delta'; \theta' \]

Extend substitution \( \theta \) and bindings \( \Delta \)
Avoid heffalump trap

Sadly, we cannot require \( \phi \) to be of form \( T \xi \), as we did before

Thread substitution through sub-patterns
Nested patterns

Three possible outcomes:

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

\[
\Gamma;\epsilon;\emptyset \vdash^p p : \sigma_1; \Delta; \theta \quad \theta(\Gamma,\Delta) \vdash \theta(u) : \theta(\sigma_2) \\
\Gamma \vdash^\alpha p \rightarrow u : \sigma_1 \rightarrow \sigma_2
\]

\text{data Term a where}
\text{Lit :: Int -> Term Int}
\text{Succ :: Term Int -> Term Int}
\text{IsZero :: Term Int -> Term Bool}
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)

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

```hs
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

Atoms \( \nu ::= x \mid C \)

Terms \( t, u ::= \nu \mid \lambda p.t \mid t u \mid t::ty \)
- \( \text{let } x = u \text{ in } t \)
- \( \text{letrec } x::ty = u \text{ in } t \)
- \( \text{case } t \text{ of } p \rightarrow t \)

Patterns \( p, q ::= x \mid C \overline{p} \)

Source types \( ty ::= a \mid ty_1 \rightarrow ty_2 \mid T \overline{ty} \)
- \( \forall \overline{a}. ty \)

Polytypes \( \sigma, \phi ::= \forall \overline{\alpha}. \tau \)

Monotypes \( \tau, \nu ::= T \overline{\tau} \mid \tau_1 \rightarrow \tau_2 \mid \alpha \mid \overline{\tau} \)

No compulsory types on binders, or on case

Type annotations on terms

Source types are part of syntax of programs

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

Atoms \( \nu ::= x | C \)

Terms \( t,u ::= \nu | \lambda p.t | tu | t::ty \\
| \text{let } x = u \text{ in } t \\
| \text{letrec } x::ty = u \text{ in } t \\
| \text{case } t \text{ of } p \rightarrow t \)

Patterns \( p,q ::= x | C \overline{p} \)

Source types \( ty ::= a | ty_1 \rightarrow ty_2 | T ty \\
| \text{forall } \overline{a}.ty \)

Polytypes \( \sigma, \phi ::= \forall \overline{\alpha}.\tau \)

Monotypes \( \tau,\nu ::= T \overline{\tau} | \tau_1 \rightarrow \tau_2 | \alpha | \top \)

---

**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

\[
\lambda x. (\text{foo } x, \text{ case } x \text{ of }
\begin{array}{l}
\text{Succ } t \rightarrow 1 \\
\text{IsZero } i \rightarrow 1 + \text{True}
\end{array}
) \quad \text{foo :: Term Int -> Bool}
\]

- Type inference guesses \(x:a56\), then \((\text{foo } x)\) forces \(a56=\text{Term Int}\), so the IsZero case can’t match
Wobbly types

\( \lambda x. (\text{foo } x, \text{case } x \text{ of } \text{Succ } t \rightarrow 1 \text{ IsZero } i \rightarrow 1 + \text{True}) \)

- 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

\[
\Gamma, (x: [\tau_1]) \vdash t : \tau_2 \\
\overline{\Gamma \vdash (\lambda x . t) : ([\tau_1] \rightarrow \tau_2)}
\]

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:

\[
\text{Unification } \vdash^u \Pi \leadsto \Theta
\]
Effect of wobbly types

- Wobbly types are impervious to a type-refining substitution

\[
\begin{align*}
\theta(\tau) &:: \tau \\
\theta(\alpha) &= \alpha & \text{if } \alpha \notin \text{dom}(\theta) \\
 &= \tau & \text{if } [\alpha \mapsto \tau] \in \theta \\
\theta(T\tau) &= T\theta(\tau) \\
\theta(\tau_1 \to \tau_2) &= \theta(\tau_1) \to \theta(\tau_2) \\
\theta(\text{Term } a) &= \tau \\
\end{align*}
\]

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

y will get a boxed type, which will not be refined
IDEA 2: directionality flag \( \delta \)

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

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

In environment $\Gamma$, term $t$ has type $\tau$

In environment $\Gamma$ and supplied context $\tau$, term $t$ is well-typed

\[
\Gamma \vdash \uparrow t : \tau
\]

\[
\Gamma \vdash \downarrow t : \tau
\]

\[
\Gamma, (x : [\tau_1]) \vdash \uparrow t : \tau_2
\]

\[
\Gamma \vdash \uparrow (\lambda x. t) : ([\tau_1] \rightarrow \tau_2)
\]

\[
\Gamma \vdash \downarrow (\lambda x. t) : (\tau_1 \rightarrow \tau_2)
\]

Guess

No guess

Local Type Inference (Pierce/Turner)
Typechecking functions

\[ \Gamma \vdash t : \tau \quad \tau = \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash u : \tau_1 \]

\[ \Gamma \vdash_\delta t u : \tau_2 \]

**Guess** the function type (probably from \( \Gamma \))

**Check** the argument type

So if \( f :: \text{Term Int} \rightarrow \text{Int} \) 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

\[ \Gamma \vdash t : \tau \quad \text{push}(\tau) = \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash u : \tau_1 \]

\[ \Gamma \vdash \delta \; t \; u : \tau_2 \]

\( \tau \) might not be an arrow type: it might wobbly!

\[
\begin{align*}
\text{push}(\tau) &::= \tau \\
\text{push}(\top \tau) &= \top \boxed{\tau} \\
\text{push}(\tau_1 \rightarrow \tau_2) &= \boxed{\tau_1} \rightarrow \boxed{\tau_2} \\
\text{push}(\bot) &= \text{push}(\tau)
\end{align*}
\]
Bore 2: guess meets check

\[ \Gamma \vdash t : \tau \quad \text{push}(\tau) = \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash u : \tau_1 \quad \vdash_\delta \tau_2 \sim \tau'_2 \]

\[ \Gamma \vdash_\delta tu : \tau'_2 \]

- **Guessing mode is easy:** \( \tau_2 = \tau'_2 \)
- **Checking mode is trickier:** \( \tau_2 \) might have different boxes than \( \tau'_2 \)

We want \( \text{strip}(\tau_2) = \text{strip}(\tau'_2) \)
Bore 2: guess meets check

\[
\begin{align*}
\Gamma \vdash t : \tau & \quad \text{push}(\tau) = \tau_1 \rightarrow \tau_2 & \quad \Gamma \vdash u : \tau_1 \quad \overset{\text{inst}\tau}{\dashv}_{\delta} \tau_2 \sim \tau_2' \\
\hline
\Gamma \vdash_{\delta} tu : \tau_2' \\
\end{align*}
\]

\[
\begin{align*}
\overset{\text{inst}\tau}{\dashv}_{\delta} \tau \sim \tau \\
\end{align*}
\]

\[
\begin{align*}
\overset{\text{inst}\tau}{\dashv} \tau \sim \tau & \quad \overset{\text{INST}\tau}{\dashv} \text{strip}(\tau) = \text{strip}(\nu) \\
\overset{\text{INST}\tau}{\dashv} \tau \sim \nu \\
\end{align*}
\]

\[
\begin{align*}
\text{strip}(\alpha) &= \alpha \\
\text{strip}(T \tau) &= T \text{strip}(\tau) \\
\text{strip}(\tau_1 \rightarrow \tau_2) &= \text{strip}(\tau_1) \rightarrow \text{strip}(\tau_2) \\
\text{strip}(\overline{\tau}) &= \text{strip}(\tau) \\
\end{align*}
\]
The good news

Just like before, modulo passing on directionality flags
Abstraction

- Lambdas use the same auxiliary judgement as case

\[
\begin{align*}
\Gamma \vdash^k \tau_1 & \quad \Gamma \vdash^a \text{p} \rightarrow \text{t} : \boxed{\tau_1} \rightarrow \tau_2 \\
\hline
\Gamma \vdash \uparrow \lambda \text{p} . \text{t} : \boxed{\tau_1} \rightarrow \tau_2 & \quad \text{ABS} \uparrow
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash^a \text{p} \rightarrow \text{t} : \tau_1 \rightarrow \tau_2 & \\
\hline
\Gamma \vdash \downarrow \lambda \text{p} . \text{t} : \tau_1 \rightarrow \tau_2 & \quad \text{ABS} \downarrow
\end{align*}
\]
Case alternatives

\[ \Gamma \vdash^a_{\delta} p \rightarrow u : \tau_1 \rightarrow \tau_2 \]

\[
\frac{
\Gamma ; \epsilon ; \emptyset \vdash^p p : \tau_1 ; \Delta ; \theta \quad \text{dom}(\Delta) \# (\text{ftv}(\tau_2)) \quad \theta(\Gamma, \Delta) \vdash_{\delta} u : \theta_{\delta}(\tau_2)
}{
\Gamma \vdash^a_{\delta} p \rightarrow u : \tau_1 \rightarrow \tau_2
}
\]

\[
\theta_{\uparrow}(\tau) = \tau
\]

\[
\theta_{\downarrow}(\tau) = \theta(\tau)
\]

Only refine result type when in checking mode
Patterns

Patterns $\Gamma;\Delta;\theta \vdash_{p} p : \tau;\Delta';\theta'$

- Bindings and type refinement from "earlier" patterns
- Augmented with bindings and type refinements from $p$
Patterns

Ensure the pattern type has the right shape

Perform wobbly unification

Same as before except...
Wobbly unification

**Goal:** $\theta$ makes the best refinement it can using only the **rigid** parts of $\Pi$

- A type is "rigid" if it has no wobbly parts.

\[
\begin{align*}
\theta(\Pi') &= \Pi \\
\text{dom}(\theta) &\neq \text{ftv}(\Pi) \\
\Pi' \text{ is rigid} &\quad \theta' \text{ is a most general unifier of } \Pi' \\
\therefore \quad \Pi &\vdash \Pi \leadsto (\theta \circ \theta') |_{\text{ftv}(\Pi)}
\end{align*}
\]
Soundness of the source

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

Theorem 4.1. If $\Gamma \vdash_{\delta} t \leadsto t' : \tau$ then $S(\Gamma) \vdash t' : S(\tau)$

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
Must $\theta$ 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".

$\theta$ is a partial unifier of $\Pi$ iff for any unifier $\Phi$ of $\Pi$, there is a substitution $\theta'$ such that: $\Phi = \theta' \circ \theta$