Compositional Explanation of Types and Algorithmic Debugging of Type Errors

Olaf Chitil
The University of York, UK

Example

```
reverse []     = []
reverse (x:xs) = reverse xs ++ x

last xs = head (reverse xs)
init    = reverse . tail . reverse

rotateR xs = last xs : init xs
```

Hugs 98:

```
Error : Type error in application
*** Expression     : last xs : init xs
*** Term           : last xs
*** Type           : [a]
*** Does not match : a
*** Because        : unification would give
                     infinite type
```

Problems:

- wrong error location
- where do the two types come from?
- scope of type variables?
- meaning of error message?

Milner's algorithm W

- introduces globally scoped type variables
- globally updates variables
- has left-right information flow

```
f (not x)  (x ++ "demo")
```

Hindley-Milner Type Inference Tree

$$\dfrac{\dfrac{\{x :: Bool\} \vdash x :: Bool \qquad \{\} \vdash not :: Bool \to Bool}{\{x :: Bool\} \vdash not\ x :: Bool} \qquad \{x :: Bool\} \vdash x :: Bool}{\{x :: Bool\} \vdash (x, not\ x) :: (Bool, Bool)}$$

- not compositional because of environment
- no proof that there exists no more general type

Solution: Principal Typings

principal type: type for given expression + type environment

$$\{x :: Bool\} \vdash x :: Bool$$

principal typing: $\underbrace{\text{type environment} + \text{type}}_{\text{typing}}$ for given expression

$$\{x :: \alpha\} \vdash x :: \alpha$$

An Inference Tree of Principal Typings is Compositional

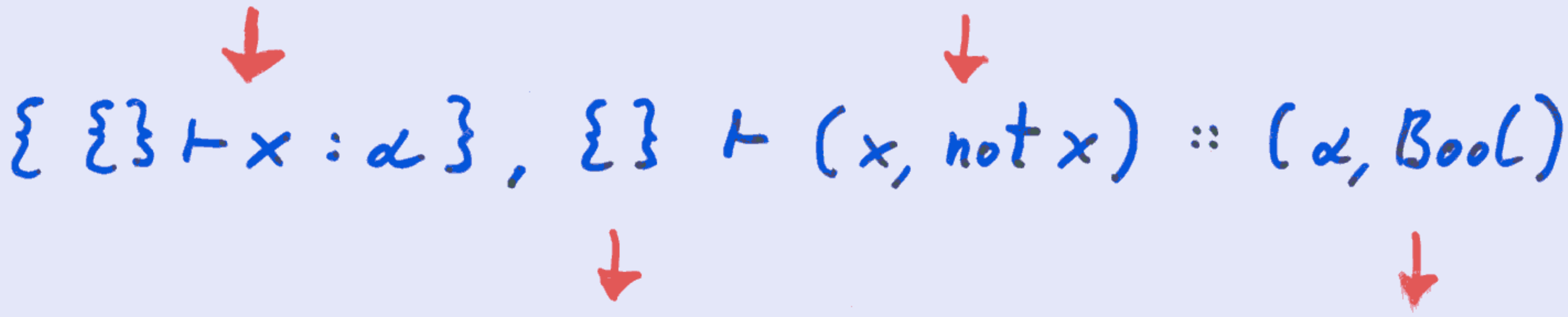
$$\dfrac{\dfrac{\{x :: \alpha\} \vdash x :: \alpha}{\{x :: Bool\} \vdash not\ x :: Bool} \qquad \{x :: \alpha\} \vdash x :: \alpha \qquad \{\} \vdash not :: Bool \to Bool}{\{x :: Bool\} \vdash (x, not\ x) :: (Bool, Bool)}$$

But x could be a let-bound, polymorphic variable

$$\{x :: \forall\alpha.\alpha\} \vdash (x, \text{not}\, x) :: (\alpha, \text{Bool})$$

the Hindley-Milner system doesn't have principal typings [Jim '96]

Solution: separate environment for let-bound variables [Mitchell '96]



The Explanation Graph

$$\dfrac{\dfrac{\{x :: \alpha\} \vdash x :: \alpha}{\{\,\{\} \vdash x :: \alpha\,\} \vdash \mathit{let}\ x = x} \qquad \dfrac{\{\} \vdash \mathit{not} :: \mathit{Bool} \to \mathit{Bool} \qquad \dfrac{\{\} \vdash x :: \alpha}{\{\} \vdash \mathit{not}\ x :: \mathit{Bool}}}{\{\} \vdash (x, \mathit{not}\ x) :: (\alpha, \mathit{Bool})}}$$

not completely syntax-directed, but compositional

Navigation through the Graph

Type error in `(last xs) : (init xs)`

because

Expressions:	`(:) (last xs)`	`init xs`
Types :	`[a] -> [a]`	`[b]`
with `xs`	`[[a]]`	`[[[b]]]`

Navigation at the Level of Polymorphic Variables

Type error in (last xs) : (init xs)

because

```
last :: [[a]] → a
init :: [[[a]]] → [a]
```

> 1

```
last :: [[a]] -> a
```

because

```
reverse :: [[a]] -> [a]
```

> 1

```
reverse :: [[a]] -> [a]
```

because

> switch to level of all program fragments

```
reverse :: [[a]] -> [a]

    because

Equation :        reverse [] = []        reverse (x:xs) = ...
with  reverse     [b] -> [c]             [[a]] -> [a]

> 2

Equation :  reverse (x:xs) = ...
with  reverse  [[a]] -> [a]
    because
Lhs/Rhs  :  reverse (x:xs)      (reverse xs) ++ x
Types    :  b                   [a]
with  reverse  [c] -> b         d -> [a]
      x        c                [a]
      xs       [c]              d

> 2
```

Expression : (reverse xs) ++ x

Type : [a]

with reverse d → [a]

x [a]

xs d

because

Expressions : (++) (reverse xs) x

Types : [b] → [b] c

with reverse a → [b]

x c

xs a

Algorithmic Debugging

Shapiro '83

Type error in : (last xs) : (init xs)

last :: [[a]] → a
Is intended type an instance ? (y/n) n

reverse :: [[a]] → [a]
Is intended type an instance ? (y/n) n

Error located! In definition of reverse
Switch to level of all program fragments

reverse :: [b] → [c]
Is intended type an instance ? (y/n) y

reverse :: [[a]] → [a]

Is intended type an instance ? (y/n) n

```
rererse (x:xs)   :: b
reverse          :: [c] -> b
x                :: c
xs               :: [c]
```

Are intended types an instance? (y/n) y

```
(reverse xs) ++ x  :: [a]
reverse            :: d -> [a]
x                  :: [a]
xs                 :: d
```

Are intended types an instance? (y/n) n

```
(++) (reverse xs)  :: [b] -> [b]
reverse            :: a -> [b]
xs                 :: a
Are intended types an instance ? (y/n)    y

Error located! Wrong expression:
(reverse xs) ++ x
```

Implementation

prototype based on Mark Jones' type checker for core Haskell

efficiency can be improved by

- piecemeal generation of explanation graph
- combination with algorithm W

Summary

- compositionality is the key to good explanations
- principal typings instead of principal types
- graph for free navigation and algorithmic debugging

Future Work

- Haskell class system
- graphical user interface