Locating the Source of Type Errors

Olaf Chitil

Programming Languages and Systems Group
The University of York
The Problem

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

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

rotateR xs = last xs : init xs

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

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

• Milner’s algorithm $\mathcal{W}$
  ▶ introduces globally scoped type variables
  ▶ globally updates variables
  ▶ has left-to-right information flow $f(\text{not } x) (x \text{ ++ } "demo")$

• A Hindley-Milner type inference tree:

\[
\begin{align*}
\{x :: \text{Bool}\} \vdash x :: \text{Bool} & \quad \{x :: \text{Bool}\} \vdash x :: \text{Bool} \\
\{\} \vdash \text{not} :: \text{Bool} \rightarrow \text{Bool} & \quad \{x :: \text{Bool}\} \vdash x :: \text{Bool} \\
\{x :: \text{Bool}\} \vdash x :: \text{Bool} & \quad \{x :: \text{Bool}\} \vdash \text{not } x :: \text{Bool} \\
\{x :: \text{Bool}\} \vdash (\text{x, not } x) :: (\text{Bool, Bool})
\end{align*}
\]

▶ not compositional because of environment
▶ no proof that there exists no more general type
Solution: Principal Typings

principal type: most general type for given expression + type environment

\( \{x :: \text{Bool}\} \vdash x :: \text{Bool} \)

principal typing: most general type environment + type for given expression typing

\( \{x :: \alpha\} \vdash x :: \alpha \)

The inference tree of principal typings is compositional:

\[
\begin{align*}
\{x :: \alpha\} & \vdash x :: \alpha \\
\{\} & \vdash \text{not} :: \text{Bool} \rightarrow \text{Bool} & \{x :: \alpha\} & \vdash x :: \alpha \\
\{x :: \text{Bool}\} & \vdash \text{not} x :: \text{Bool} & \left[\text{Bool}/\alpha\right] \\
\{x :: \text{Bool}\} & \vdash (x, \text{not} x) :: (\text{Bool, Bool}) & \left[\text{Bool}/\alpha\right]
\end{align*}
\]
An Obstacle

**But** $x$ could be a let-bound, polymorphic variable.

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

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

**Solution:** separate environments for let-bound variables [Mitchell ’96].

$$\Downarrow \quad \Downarrow$$

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

$$\Downarrow \quad \Downarrow$$
Polymorphic type environment still creates global dependencies
⇒ “copy” inference tree of definition to use occurrences.

Not completely syntax-directed, but compositional.
Navigation through the Explanation Graph

Explanation at expression level:

**Error**: unification would lead to infinite type
in expression: `(last xs) : (init xs)`

  because

Expression: `(:) (last xs)` \hspace{2cm} `init xs`
Type: \hspace{2cm} `[a]->[a]` \hspace{2cm} `[a]`
with `xs` \hspace{2cm} `[[a]]` \hspace{2cm} `[[[a]]]`

Explanation at function level:

**Error**: unification would lead to infinite type
in expression: `(last xs) : (init xs)`

  because

Expression: `last` \hspace{2cm} `init`
Type: \hspace{2cm} `[[a]]->a` \hspace{2cm} `[[[a]]]->[a]`
Algorithmic Debugging

Shapiro ’83

\[
\begin{array}{cccccccc}
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\cdots & \checkmark & \cdots & \checkmark & \cdots & \cdots & \cdots & \cdots \\
\cdots & \checkmark & \cdots & \times & \cdots & \cdots & \cdots & \cdots \\
\cdots & \times & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\cdots & \times & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\cdots & \times & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\end{array}
\]
Algorithmic Debugging of the Example I

Error: unification would lead to infinite type
in expression: (last xs) : (init xs)

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

head     ::= [a]->a
Is intended type an instance? (y/n) y

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

ERROR LOCATED! Wrong definition of:
reverse  ::= [[a]]->[a]

Switch to detailed level of program fragments.
Algorithmic Debugging of the Example II

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

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

(reverse xs) ++ x :: [b]
reverse :: a -> [b]
xs :: a
x :: [b]
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 program fragment:
(reverse xs) ++ x
Locating the Source of Type Errors

Summary

- compositionality is the key to meaningful explanations
- principal typings instead of principal types
- interactive free navigation and algorithmic debugging

Future Work

- refine method
  e.g. quick navigation to explanation of marked type constructor
- combine with other methods
  e.g. minimal unsolvable constraints of Haack & Wells
  and soft typing of Neubauer & Thiemann
- implement for full Haskell
  including source browser showing typing of any marked expression
Debugging Haskell Programs with the Haskell Tracer Hat

Joint work with Colin Runciman and Malcolm Wallace.

Future Work

- formally relate operational semantics and trace
  ⇒ better understanding e.g. of trusting
- new views: animation à la GHood, locating black holes, stories of Booth
- tracing the functional-logic language Curry
  (with Michael Hanus and Frank Huch at Kiel)
- tracing type inference
Programming and Programming Languages

New Language Constructs
- future work: relationship
- future work: free theorems
- PhD: deforestation
- locating the source of type errors
- future work: refactoring for ADTs

Operational Semantics
- MSc: strict & non-strict

Program Transformation
- Haskell tracer

Programming Tools