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

- **A Hindley-Milner type inference tree:**

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

  - ▷ **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

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

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

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

The inference tree of principal typings is compositional:

$$\dfrac{\{\mathbf{x} :: \alpha\} \vdash \mathbf{x} :: \alpha \qquad \dfrac{\{\} \vdash \mathbf{not} :: \mathbf{Bool} \rightarrow \mathbf{Bool} \qquad \{\mathbf{x} :: \alpha\} \vdash \mathbf{x} :: \alpha}{\{\mathbf{x} :: \mathbf{Bool}\} \vdash \mathbf{not\ x} :: \mathbf{Bool}}\ [\mathbf{Bool}/\alpha]}{\{\mathbf{x} :: \mathbf{Bool}\} \vdash (\mathbf{x}, \mathbf{not\ x}) :: (\mathbf{Bool}, \mathbf{Bool})}\ [\mathbf{Bool}/\alpha]$$

An Obstacle

But $\mathbf{x}$ **could be a let-bound, polymorphic variable.**

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

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

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

$$\Downarrow \qquad\qquad\qquad\qquad \Downarrow$$

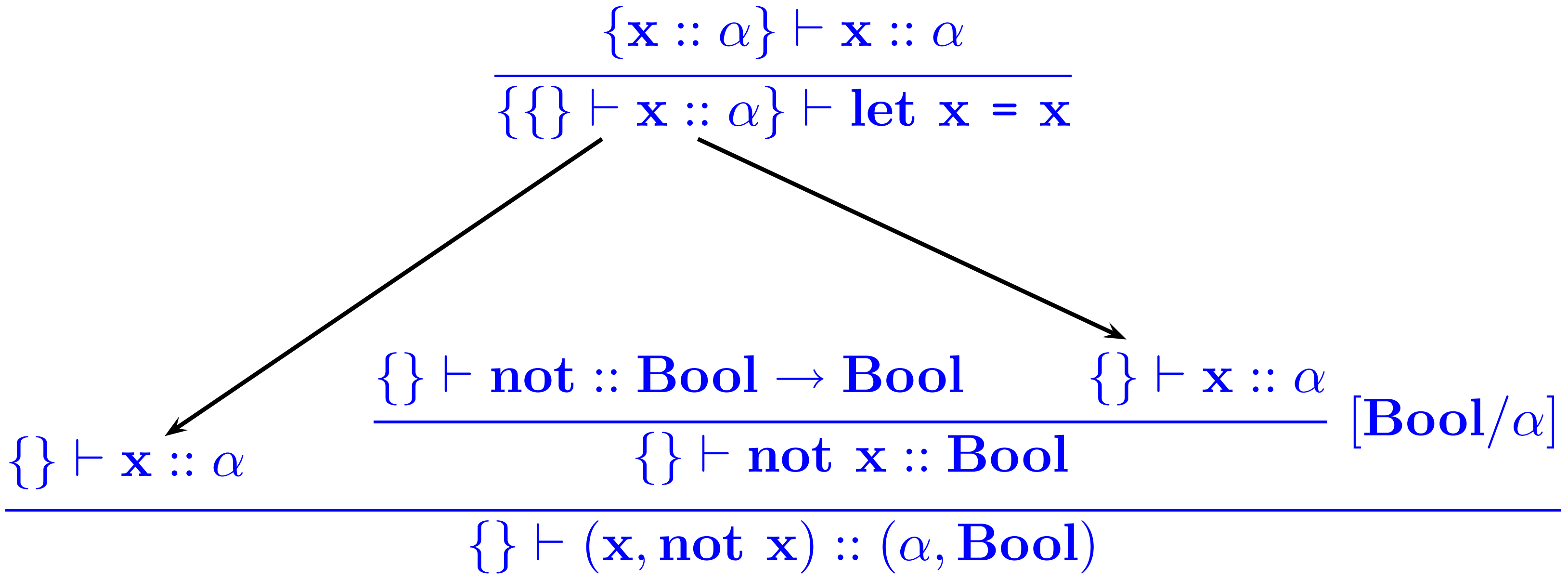
$$\{\{\} \vdash \mathbf{x} :: \alpha\}, \{\} \vdash (\mathbf{x}, \mathbf{not}\ \mathbf{x}) :: (\alpha, \mathbf{Bool})$$

$$\Downarrow \qquad\qquad\qquad\qquad \Downarrow$$

The Explanation Graph

Polymorphic type environment still creates global dependencies

⇒ "copy" inference tree of definition to use occurrences.



$$\frac{\{\mathbf{x} :: \alpha\} \vdash \mathbf{x} :: \alpha}{\{\{\} \vdash \mathbf{x} :: \alpha\} \vdash \mathbf{let\ x\ =\ x}}$$

$$\frac{\{\} \vdash \mathbf{x} :: \alpha \qquad \dfrac{\{\} \vdash \mathbf{not} :: \mathbf{Bool} \to \mathbf{Bool} \qquad \{\} \vdash \mathbf{x} :: \alpha}{\{\} \vdash \mathbf{not\ x} :: \mathbf{Bool}}\,[\mathbf{Bool}/\alpha]}{\{\} \vdash (\mathbf{x}, \mathbf{not\ x}) :: (\alpha, \mathbf{Bool})}$$

Not completely syntax-directed, but compositional.

Navigation through the Explanation Graph

Explanation at expression level:

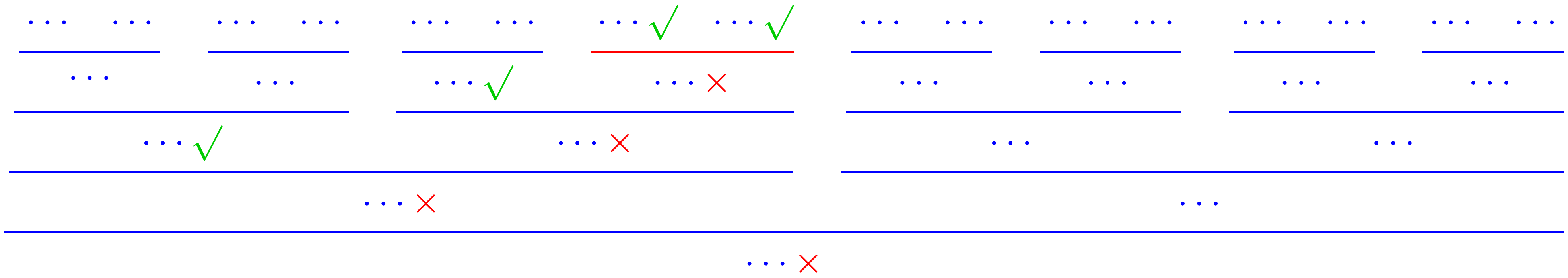
```
Error: unification would lead to infinite type
in expression: (last xs) : (init xs)
  because
Expression: (:) (last xs)                 init xs
Type:       [a]->[a]                      [a]
with xs     [[a]]                         [[[a]]]
```

Explanation at function level:

```
Error: unification would lead to infinite type
in expression: (last xs) : (init xs)
  because
Expression: last                          init
Type:       [[a]]->a                      [[[a]]]->[a]
```

Algorithmic Debugging

Shapiro ’83



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

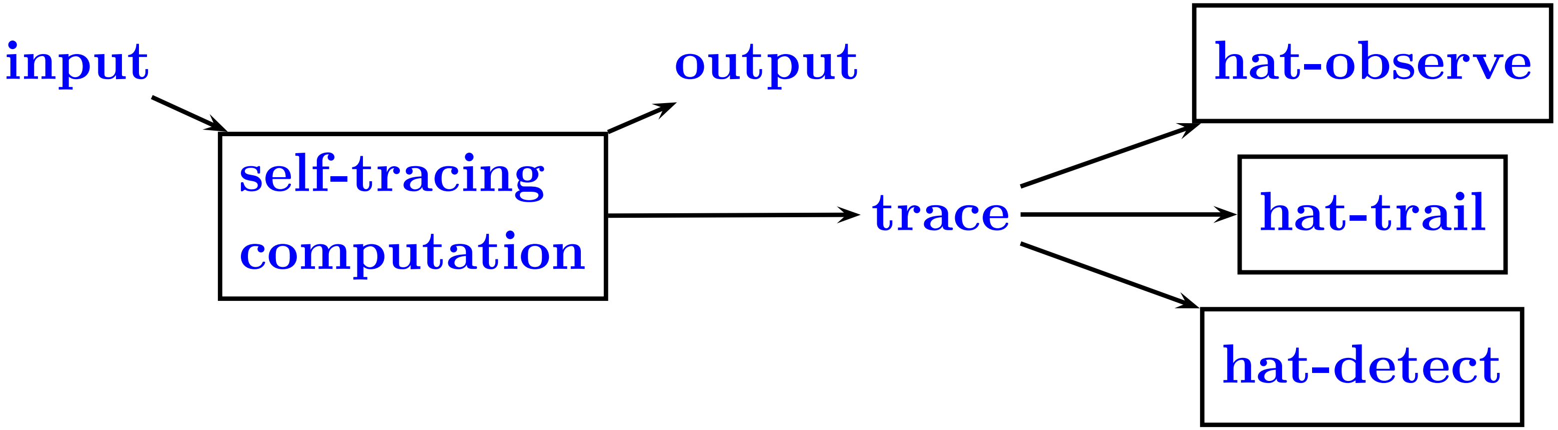
- compositionality is the key to meaningful explanations
- principal typings instead of principal types
- interactive free navigation and algorithmic debugging

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

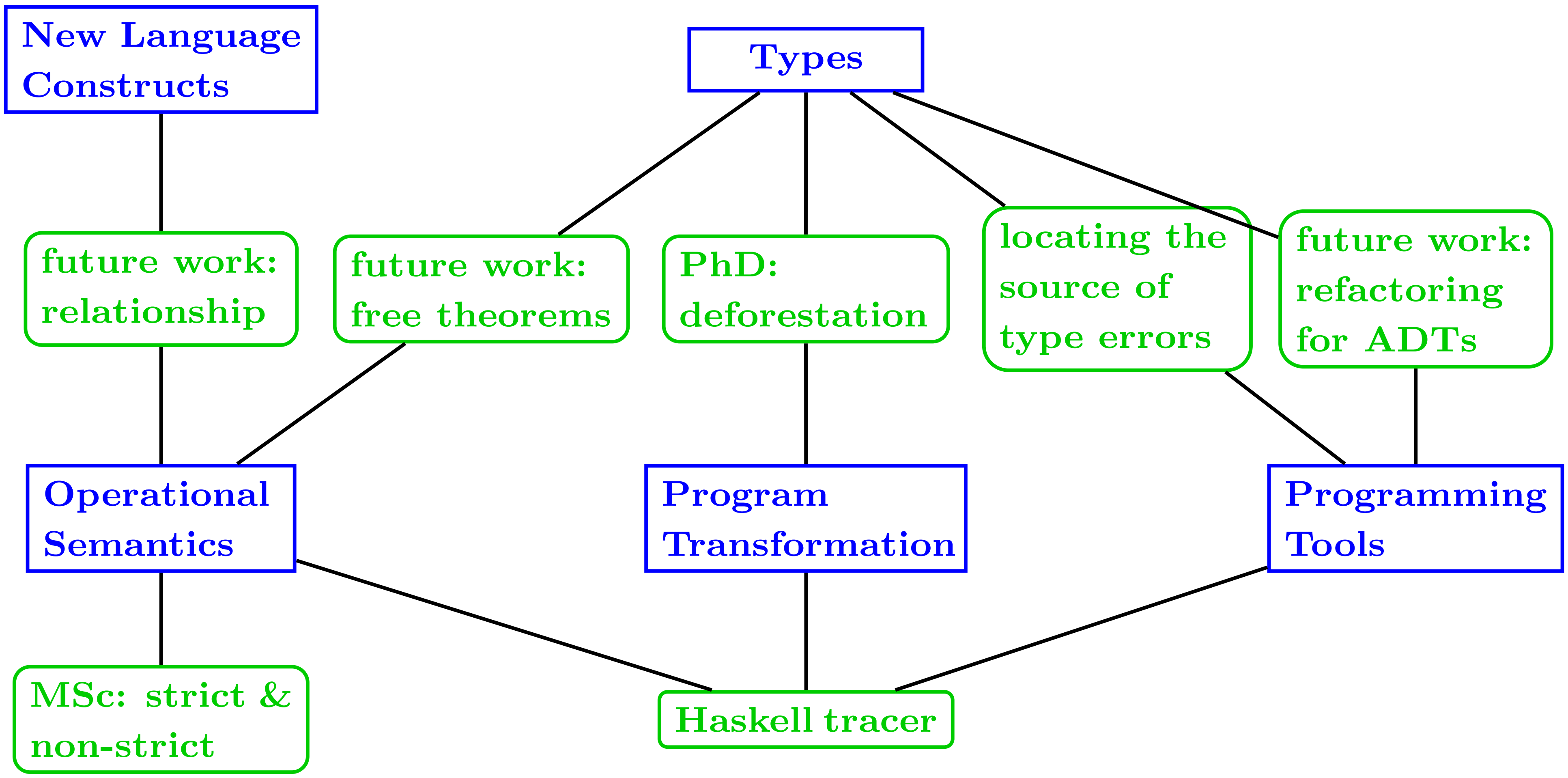
Joint work with Colin Runciman and Malcolm Wallace.



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

Types

future work: relationship

future work: free theorems

PhD: deforestation

locating the source of type errors

future work: refactoring for ADTs

Operational Semantics

Program Transformation

Programming Tools

MSc: strict & non-strict

Haskell tracer