Werkzeuge zur Lokalisierung der Ursachen von Typfehlern in Programmen

Olaf Chitil

The University of York

United Kingdom

Das Hindley-Milner Typsystem

Grundlage der Typsysteme von ML, Haskell, Clean, Curry, ...

- **garantiert Nichtauftreten einer großen Klasse von Laufzeitfehlern**
- **flexibel und ausdrucksstark durch parametrische Polymorphie**

```
(:) :: a -> [a] -> [a]
(++) :: [a] -> [a] -> [a]
concat :: [[a]] -> [a]
map :: (a -> b) -> [a] -> [b]
```

- **benötigt keine Typannotationen**

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

Das Problem

```
perms :: [a] -> [[a]]
perms [] = [[]]
perms (x:xs) = concat (map addX (perms xs))
  where
  addX [] = [[x]]
  addX (y:ys) = (x:y:ys) : map (y ++) (addX ys)
```

Hugs:

```
ERROR Test.hs:77
- Type error in application
*** Expression     : x : y : ys
*** Term           : x
*** Type           : a
*** Does not match : [a]
*** Because        : unification would
                     give infinite type
```

Glasgow Haskell Compiler:

```
Test.hs:77:
Cannot unify the type-signature
variable ‘a’ with the type ‘[a1]’
  Expected type: [a1]
  Inferred type: a
In the first argument of ‘(++)’, namely ‘y’
In the first argument of ‘map’,
namely ‘(y ++)’
```

- **Gegebener Ausdruck ist nicht Fehlerursache**
- **Bedeutung der Typen? (woher? Gültigkeitsbereich der Typvariablen?)**

Typinferenz

- **jedes Programmkonstrukt stellt Bedingungen an Typen:**
  Constraints

  `f x` $\Rightarrow$ $t_f = t_1 \texttt{->} t_2$ **und** $t_x = t_1$

- **Typinferenz besteht aus dem Lösen aller Constraints eines Programms**
- **Constraints unlösbar $\Rightarrow$ Typfehler**

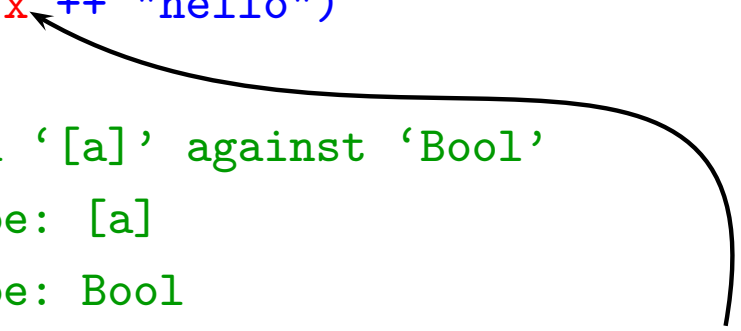
Ursache des Problems

Verwendung von Milner's Typinferenzalgorithmus $\mathcal{W}$ oder Variante $\mathcal{M}$.

- **führt globale Typinferenzvariablen ein**
- **löst Constraints inkrementell während Syntaxbaumdurchlaufs**

```
g x = (not x, x ++ "hello")

Couldn't match `[a]' against `Bool'
  Expected type: [a]
  Inferred type: Bool
In the first argument of `(++)', namely `x'
In the definition of `g': (not x, x ++ "hello")
```

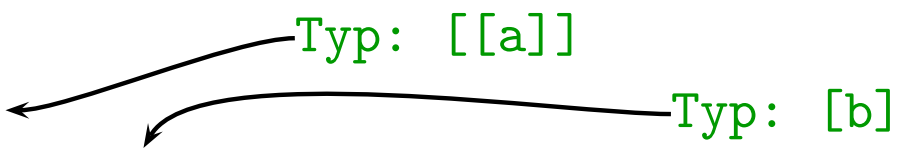


Fehlermeldung wird aus aktuellem Zustand generiert

- **meldet Ausdruck an dem Constraints sich als unlösbar erweisen**
- **angegebene Typen sind Vorstufen eines endgültigen Typs**
- **Typen enthalten Typinferenzvariablen und Typsignaturvariablen**

Probleme eines Typ-Browsers

```
perms :: [a] -> [[a]]
perms [] = [[]]
perms (x:xs) = concat (map addX (perms xs))
  where
  addX [] = [[x]]
  addX (y:ys) = (x:y:ys) : map (y++) (addX ys)
```



Gültigkeitsbereich einer Typvariable geht über einen Typ hinaus

- **wie soll Browser Gültigkeitsbereich angeben?**
- **Typ alleine nicht verständlich**

Bei Typfehler

- **ist für Teilausdruck nur Teil des Typs inferiert worden**

Standpunkt

- **Die Ursache eines Typfehlers kann nicht automatisch lokalisiert werden. Der Computer kennt nicht die Intentionen des Programmierers.**
- **Der Programmierer benötigt eine Erklärung die**
  - ▷ **nachvollziehbar ist und**
  - ▷ **ihn nicht in Informationen ertränkt.**

Daher:

- **Eine Erklärung von Typen und Fehlern muß interaktiv sein, damit**
  - ▷ **der Programmierer seine Intentionen einbringen kann und**
  - ▷ **die relevanten Erklärungsteile auswählen kann.**
- **Eine Erklärung muß kompositionell sein, d.h.**
  - ▷ **eine Typangabe ergibt sich aus den Typangaben von Komponenten,**
  - ▷ **eine Typangabe ist einfach verständlich.**

Ein erster Schritt

Hindley-Milner Typregeln

$$\frac{\mathbf{U} \vdash \mathbf{M} :: \mathbf{t_1} \to \mathbf{t_2} \quad \mathbf{U} \vdash \mathbf{N} :: \mathbf{t_1}}{\mathbf{U} \vdash \mathbf{M\,N} :: \mathbf{t_2}} \qquad \{\mathbf{x} :: \mathbf{t}, \ldots\} \vdash \mathbf{x} :: \mathbf{t}$$

beschreiben Typinferenzbaum eines Programms

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

- **nicht kompositionell wegen der Typumgebung U**
- **kein Beweis, daß nicht ein allgemeinerer Typ existiert**

Lösung: Allgemeinste Typungen

allgemeinster Typ: **Typ für gegebenen Ausdruck + Typumgebung**

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

allgemeinste Typung: $\underbrace{\textbf{Typumgebung + Typ}}_{\textbf{Typung}}$ **für gegebenen Ausdruck**

$$\{x :: a\} \vdash x :: a$$

Der Typinferenzbaum für allgemeinste Typungen ist kompositionell:

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

Ein Hindernis

Aber x könnte einen polymorphen Typ haben.

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

Das Hindley-Milner System hat keine allgemeinsten Typungen [Jim '96, Wells '02].

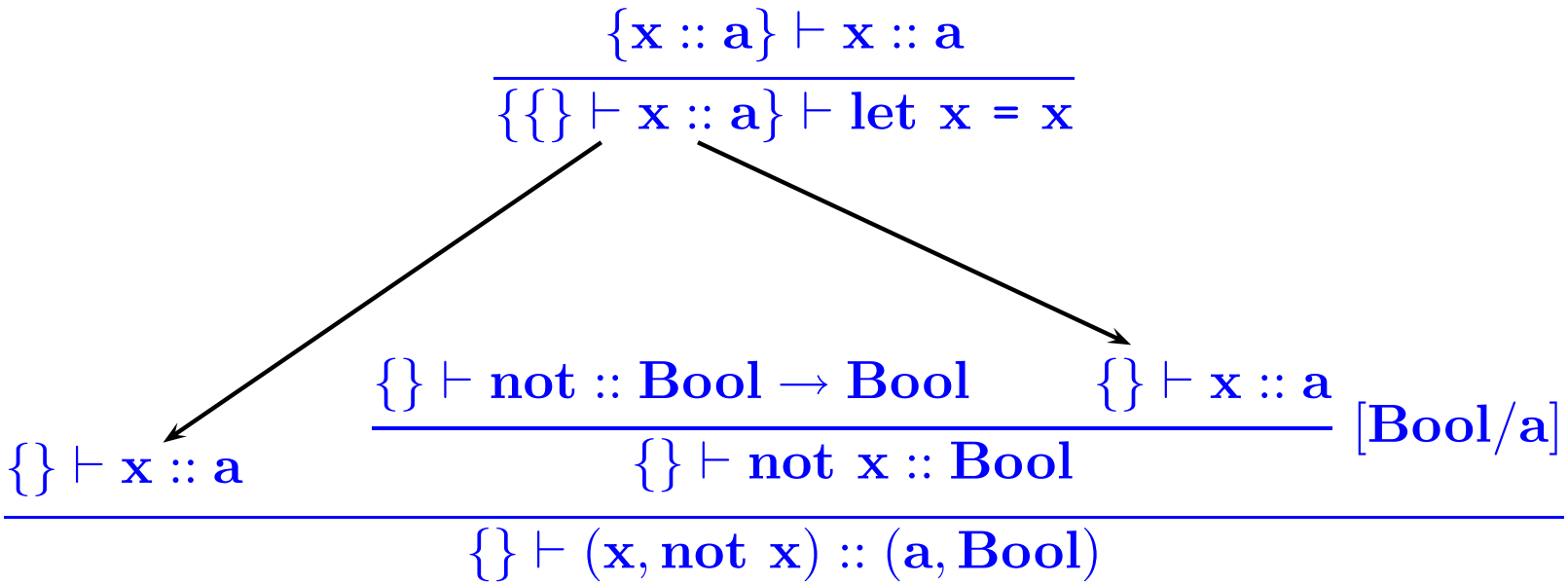
Lösung: **getrennte Typumgebung für polymorphe Variablen (durch Gleichungen definierte Variablen vs. Argumentvariablen) [Mitchell '96].**

$$\begin{array}{c} \Downarrow \qquad\qquad \Downarrow \\ \{\{\} \vdash \mathbf{x} :: \mathbf{a}\}, \{\} \vdash (\mathbf{x}, \mathbf{not\ x}) :: (\mathbf{a}, \mathbf{Bool}) \\ \Downarrow \qquad\qquad\qquad\qquad \Downarrow \end{array}$$

Der Erklärungsgraph

Polymorphe Typumgebung erzeugt globale Abhängigkeiten.

$\Rightarrow$ "kopiere" Inferenzbaum der Definition zu jeder Verwendungstelle.



Nicht vollständig syntaxgerichtet, aber kompositionell.

Fehlermeldung als Konflikt zweier Typungen

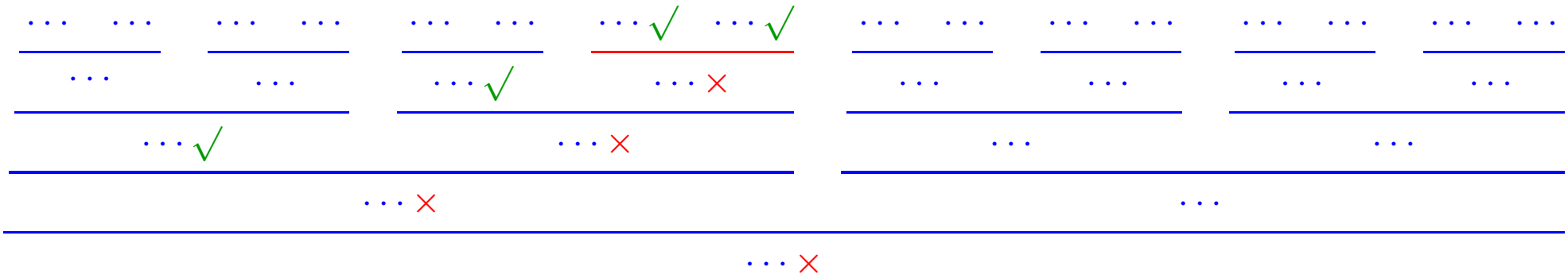
```
Error: unification would lead to infinite type
in expression: (x : (y : ys)) : ((map ((++) y)) (addX ys))
  because
Expression: (:) (x : (y : ys))          (map ((++) y)) (addX ys)
Type:       [[a]]->[[a]]                [[b]]
with x      a
     y      a                           [b]
     ys     [a]                         c
     addX                               c->[[b]]
```

Navigation durch den Erklärungsgraph

```
Expression: (map ((++) y)) (addX ys)
Type:       [[a]]
with y      [a]
     addX   b->[[a]]
     ys     b
  because
Expression: map ((++) y)                addX ys
Type:       [[c]]->[[c]]                e
with y      [c]
     addX                               d->e
     ys                                 d
```

Algorithmisches Debuggen

Shapiro ’83



Algorithmisches Debuggen des Beispiels

```
(:) (x : (y : ys)) :: [[a]]->[[a]]
x                  :: a
y                  :: a
ys                 :: [a]
Are intended types an instance? (y/n) y

(map ((++) y)) (addX ys) :: [[a]]
y                        :: [a]
addX                     :: b->[[a]]
ys                       :: b
Are intended types an instance? (y/n) n

map ((++) y) :: [[a]]->[[a]]
y            :: [a]
Are intended types an instance? (y/n) n

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

ERROR LOCATED! Wrong program fragment: (++) y :: [a]->[a]
```

Beispiel zum Algorithmischen Debuggen auf zwei Ebenen

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

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

rotateR xs = last xs : init xs

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

Lokalisiere erst fehlerhafte Funktionsdefinition,
dann fehlerhaften Ausdruck.

Algorithmisches Debuggen 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.
```

Algorithmisches Debuggen 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
```

Erweiterung um Klassen

Zum systematischen Überladen von Funktionen verwendet Haskell Klassen.

```
class Eq a where
  (==) :: a -> a -> Bool


elem :: Eq a => a -> [a] -> Bool


42 :: Num a => a
```

Beispiel:

```
class1 = (print . div) 42
```

Hugs:

```
ERROR Test.hs:1 - Unresolved top-level overloading
*** Binding                : class1
*** Outstanding context : (Show (b -> b), Integral b)
```

Ein anderer Ansatz: Helium

- **vereinfachte Sprache und Compiler zum Erlernen von Haskell**
- **Typinferenz in 2 Phasen:**
  Generieren von Constraints und Lösen von Constraints
- **kann fehlerhafte Programme automatisch archivieren**
- **Heuristiken zum Aufzeigen möglicher Fehlerursachen**
- **man kann spezialisierte Typregeln und Fehlermeldungen definieren**

```
(77,28): Type error in application
 expression       : map (y ++) (addX ys)
 term             : map
   type           : (a   -> b  ) -> [a    ] -> [b    ]
   does not match : ([c] -> [c]) -> [[[c]]] -> [[[c]]]
 because          : unification would give infinite type
```

Ein anderer Ansatz: Haack & Wells

- **separieren ebenfalls Constraint-Generierung und -Lösen**
- **bestimmen eine minimale unlösbare Menge von Constraints**
- **diese bestimmt einen Programmteil (slice) als Fehlerursache**

```
perms :: [a] -> [[a]]
perms [] = [[]]
perms (x:xs) = concat (map addX (perms xs))
  where
  addX [] = [[x]]
  addX (y:ys) = (x:y:ys) : map (y ++) (addX ys)
```

Ein anderer Ansatz: Neubauer & Thiemann

- **Erweiterung um Summentypen macht jedes Programm typbar**

  ```
  f :: Int -> (Bool | Maybe Int)
  ```

  ⇒ Unabhängigkeit vom Typinferenzalgorithmus

- **Summe mehrerer Typkonstruktoren zeigt Typfehler auf**

- **Datenflußannotationen helfen bei der Ursachensuche**

```
perms :: [a] -> [[a]]
perms [] = [[]]
perms (x:xs) = concat (map addX (perms xs))
  where
  addX [] = [[x]]
  addX (y:ys) = (x:y:ys) : map (y ++) (addX ys)
```

```
[b] -> [[b]] where b = [b] | [a]
```

Lokalisierung der Ursachen von Typfehlern

Zusammenfassung

- **Kompositionalität ist der Schlüssel zu verständlichen Erklärungen**
- **allgemeinste Typungen statt allgemeinster Typen**
- **interaktive freie Navigation und Algorithmisches Debuggen**

Geplant

- **Quellcode-Browser für Typungen**
- **Verfeinerung der Methode**
  z. B. Sprung zur Erklärung eines markierten Typkonstruktors
- **Kombination mit anderen Methoden**
- **Werkzeug für ganz Haskell**