Typüberprüfung/Typinferenz

...von zentraler Bedeutung für die Implementierung (nicht nur) funktionaler Programmiersprachen wie Haskell

Wir unterscheiden...

- monomorphe und
- polymorphe

Typüberprüfung/Typinferenz.

Typüberprüfung/Typinferenz

Wichtig ebenfalls in diesem Zusammenhang...

- Überladen (als Spezialfall von Polymorphie (ad-hoc Polymorphie)
- Typklassen (wie etwa `Num`, `Eq`, etc.)

Literaturhinweis

Die folgende Darstellung orientiert sich an...

- Kapitel 13
  Simon Thompson. *Haskell – The Craft of Functional Programming*, Addison-Wesley, 2nd edition, 1999.
- Kapitel 5
  Martin Erwig. *Grundlagen funktionaler Programmierung*. Oldenbourg Verlag, 1999.

Typisierte Programmiersprachen

Wir unterscheiden...

- Programmiersprachen mit...
  - *schwacher*
  - *starker*

  Typung

Vorteile

...der Benutzung getypter Programmiersprachen

- *Verläßlicherer Code*:
  ...viele Programmierfehler können schon zur Übersetzungszeit aufgedeckt werden; Typkorrektheit ist Korrektheitsbeweis auf Abstraktionsniveau der Typen
- *Effizienterer Code*: ...Typüberprüfungen zur Laufzeit sind unnötig
- *Effektivere Programmentwicklung*:
  ...Typinformation stellt zusätzliche Programmdokumentation dar
  ...das *Verstehen*, *Weiterentwickeln* und *Warten* von Programmen wird vereinfacht, wie auch die Suche nach vordefinierten Bibliotheksfunktionen ("gibt es eine Bibliotheksfunktion, die Duplikate aus einer Liste entfernt, angesetzt auf `[2,3,2,1,3,4]` die Ausgabe `[2,3,1,4]` liefert? Suche nach einer Funktion mit dem Typ `(Eq a) => [a] -> [a]`".)

Worum geht es? 1(4)

Ausdrücke in Haskell haben alle einen definierten Typ.

Dieser kann...

- monomorph
- polymorph
  - durch Typklassenbedingungen eingeschränkt

...und *explizit* angegeben sein wie in folgenden Beispielen:

```
'w'  :: Char                            -- monomorph
flip :: (a -> b -> c) -> (b -> a -> c)  -- polymorph
elem :: Eq a => a -> [a] -> Bool        -- polymorph mit
                                           Typklassenbedingung
```


Worum geht es? 2(4)

Der Typ kann von Haskell-Übersetzern und -Interpretierern aber auch automatisch erschlossen werden wie in folgendem Beispiel:

```
magicType = let
              pair x y z = z x y
              f y = pair y y
              g y = f (f y)
              h y = g (g y)
              in h (\x->x)
```


Worum geht es? 3(4)

Der Aufruf von `:t magicType` in Hugs liefert:

```
Main> :t magicType
magicType ::
  (((((((a -> a) -> (a -> a) -> b) -> b) ->
  (((a -> a) -> (a -> a) -> b) -> b) -> c) -> c) ->
  (((((a -> a) -> (a -> a) -> b) -> b) ->
  (((a -> a) -> (a -> a) -> b) -> b) -> c) -> c) -> d) -> d) ->
  ((((((a -> a) -> (a -> a) -> b) -> b) -> (((a -> a) ->
  (a -> a) -> b) -> b) -> c) -> c) -> (((((a -> a) ->
  (a -> a) -> b) -> b) -> (((a -> a) ->
  (a -> a) -> b) -> b) -> c) -> c) -> d) -> d) -> e) -> e
```

Worum geht es? 4(4)

Zentrale Frage...

- Wie gelingt Hugs, diese Typinformation automatisch zu erschließen?

Ihre systematische Beantwortung führt uns auf Begriffe wie...

- *Typanalyse/-überprüfung*
- *Typsysteme* und
- *Typinferenz*

Zunächst eine informelle, beispielgetriebene Annäherung...

Monomorphe Typüberprüfung

Wir betrachten vereinfachend zunächst eine Situation ohne...

- Polymorphie

Kennzeichnend für diese Situation...

- Ein Ausdruck hat
  - entweder genau einen Typ (und ist dann wohlgetypt)
  - oder keinen (und ist dann nicht wohlgetypt)

Vereinbarung:
Die Polymorphie polymorpher oder überladener Funktionen wird (duch Indizierung) explizit aufgelöst wie in...

```
+_{Int} :: Int -> Int -> Int
length_{Char} :: [Char] -> Int
```


Typanalyse

Zentrale Idee...

- Auswertung des Anwendungskontextes, in den ein Ausdruck eingebettet ist

Im folgenden dazu eine Reihe von Beispielen...

Typinferenz bei Funktionsapplikationen

Der Kontext von Funktionsanwendungen erlaubt folgende Rückschlüsse auf die involvierten Typen...

```
f muss Funktionstyp haben...          e muss Typ s haben...
       s -> t                                   s
          \                                    /
           \___________        ____________/
                       \       /
                        \     /
                        \/   \/
                          f e
                          /\
                          /
                         /
f e muss (Resultat-) Typ t haben...
t
```


Typinferenz bei anderen Ausdrücken 1(2)

Korrekt getypt...

```
Char -> Int     Char
          \      \
           \      \
           \/     \/
            ord   'c'   +_{Int}    3_{Int}
            /\            /\         /\
             |            |          |
             |            |          |
            Int   Int -> Int -> Int   Int
```


Typinferenz bei Ausdrücken 2(2)

Fehlerhaft getypt...

```
Char -> Int     Char
          \      \
           \      \
           \/     \/
            ord   'c'   +_{Int}      False
            /\            /\          /\
             |            |           |
             |            |           |
            Int   Int -> Int -> Int   Bool

            /\    /\    /\           /\
             |_____|     |____________|
                |               |
                |               |
        Erwarteter und     Erwartet ...Int
        uebergebener       Uebergeben ...Bool
        Argumenttyp        => TYPFEHLER
        stimmmen ueberein
        => TYPKORREKT
```


Typinferenz bei Funktionsdefinitionen

```
f :: t1 -> t2 -> ... -> tk -> t
f a1 a2 ... ak
  | b1   = e1
  | b2   = e2
  ...
  | bk   = ek
```

Drei Dinge sind zu überprüfen...

- Jeder Wächter `bi` muss vom Typ `Bool` sein
- Der Rückgabewert jeden Ausdrucks `ei` muss vom Typ `t` sein
- Das Muster jeden Arguments `pi` muss mit dem Typ des formalen Parameters konsistent sein, also mit dem Typ `ti`.

Konsistenz von Mustern 1(2)

Informell:
Ein Muster ist *konsistent* mit einem Typ, wenn es zu den Werten des Typs "passt".

Im einzelnen:

- Eine Variable ist konsistent mit jedem Typ
- Ein Literal ist konsistent zu seinem Typ
- Ein Muster `p:q` ist konsistent mit dem Typ `[t]`, wenn `p` konsistent mit dem Typ `t` und `q` konsistent mit dem Typ `[t]` ist
- ...

Konsistenz von Mustern 2(2)

Beipiel:

- `[42:xs]` ...ist konsistent mit `[Int]`
- `[x:xs]` ...ist konsistent mit jedem Typ

Polymorphe Typüberprüfung 1(2)

Kennzeichnend für die Situation polymorpher Typüberprüfung...

- Ein Ausdruck kann
  - mehrere Typen haben (und wohlgetypt sein)

Zentral für die Lösung des polymorphen Typüberprüfungsproblems...

- *Constraint satisfaction* ("Randbedingungserfüllung")
- *Unifikation*

Polymorphe Typüberprüfung 2(2)

Betrachte...

```
length :: [a] -> Int
```

Informelle Interpretation...

```
length :: [a] -> Int
```

...als Abkürzung für

```
length :: [t] -> Int
```

wobei `t` beliebiger monomorpher Typ ist, also Abkürzung für...

```
[Int] -> Int
[(Bool,Char)] -> Int
...
```


Polymorphe Typüberprüfung – Beispielsfortsetzung

Im Beispiel von...

```
length ['c','a']
```

lässt sich aus dem Aufrufkontext auf den Typ...

```
length :: Char -> Int
```

...schließen

Beobachtung

Die vorigen Beispiele erlauben folgende Schlussfolgerung...

Die Anwendungskontexte von Ausdrücken erlegen

- ...dem Ausdruckstyp unterschiedliche Randbedingungen (engl. constraints) auf.

Damit reduziert sich Typüberprüfung auf das Problem, ob...

- Typen so für die verschiedenen Ausdrücke bestimmt werden können, dass alle Randbedingungen erfüllt sind.

Weitere Beispiele – Beispiel 1

Betrachte:

```
f (x,y) = (x , ['a' .. y])
```

Beobachtung:

- `f` ...erwartet als Argumente Paare, wobei gilt
  - 1. Paarkomponente ...unterliegt keinen einschränkenden Randbedingungen
  - 2. Paarkomponente ...`y` muss vom Typ `Char` sein, da in Ausschnittstypelement `['a' .. y]` benutzt

Schlussfolgerung für Typ von `f` somit...

```
f :: (a , Char) -> (a , [Char])
```


Beispiel 2

Betrachte:

```
g (m,zs) = m + length zs
```

Beobachtung:

- `g` ...erwartet als Argumente Paare, wobei gilt
  - 1. Paarkomponente ...`m` muss von numerischem Typ sein, da als Operand von `+` benutzt
  - 2. Paarkomponente ...`zs` muss vom Typ `[b]` sein, da als Argument der Funktion `length` mit `length :: [b] -> Int` benutzt

Schlussfolgerung für Typ von `g` somit...

```
g :: (Int, [b]) -> Int
```


Beispiel 3 1(2)

Betrachte die Komposition der beiden vorhergehenden Beispiele:

```
g . f
```

Beobachtung:

- `g . f` ...in einer Komposition `g . f` wird das Resultat von `f` zur Eingabe von `g`
  In diesem Beispiel somit...
  - Resulttyp `f` ...`(a , [Char])`
  - Argumenttyp `g` ...`(Int , [b])`
  - Gesucht ...Typen für `a` und `b`, die die obigen beiden Randbedingungen erfüllen

Beispiel 3 2(2)

Veranschaulichung:

```
                  g . f
                 /\    /\
                /        \
               /          \
  (Int, [b]) -> Int        (a , Char) -> (a , [Char])
       /\                                      /\
        \                                     /
         \                                   /
          \                                 /
       Die Eingabe von g                   /
                   ist die Ausgabe von f
```

Schlussfolgerung für Typ von `g . f` somit...

```
g . f :: (Int, [Char]) -> Int
```


Unifikation im Beispiel

Entscheidend für die Schlussfolgerung im vorigen Beispiel...

- Unifikation

Veranschaulichung:

```
                    --------------------------------------
 ___________________|_______________                      |
|                   |               |    (Int,[Int])      |
|  (Bool,Char)      |               |                     |
|                   |               |    (Int,[[c]])      |
|  (c->c,Char)      |  (Int,[Char]) |                     |
|                   |               |    ...              |
|  ...              |               |                     |
|                   --------------------------------------
|                                   |
|___________________________________|

    (a,[Char])                          (Int,[b])
```

Unifikation

Sprechweisen...

- *Instanz* eines Typausdrucks ...entsteht durch Einsetzen eines konkreten Typausdrucks für eine Typvariable
- *Gemeinsame Instanz* zweier Typausdrücke ...wenn Typinstanz jedes der beiden Ausdrücke

Unifikationsproblem...

- Suche nach der *allgemeinsten gemeinsamen Typinstanz* (engl. *most general common instance*)

Im vorigen Beispiel:

- Allgemeinste gemeinsame Typinstanz von `(a,Char)` und `(Int,[b])`
  ...`(Int, [Char])`

Weiteres über Unifikation 1(3)

Unifikation resultiert i.a. nicht in eindeutigen Typen...

Beispiel:

```
  (a,[a])        ([b],c)
        \        /
         \      /
         \/    \/
        ([b],[[b]])
most general common instance
```

Beobachtung:

- `(a,[a])` ...erfordert, dass die zweite Komponente Liste von Elementen vom Typ der ersten Komponente ist
- `([b],c)` ...erfordert, dass die erste Komponente von einem Listentyp ist
- Zusammen ...der allgemeinste gemeinsame Typ von `(a,[a])` und `([b],c)` ist `([b],[[b]])`

Weiteres über Unifikation 2(3)

Beachte:

- `([Bool],[[Bool]])`, `([[c]],[[[c]]])` sind...
  - Instanzen von `([b],[[b]])`
  - aber keine "Vereinheitlicher"(engl. unifier): `([b],[[b]])` ist keine Instanz von beiden

Weiteres über Unifikation 3(3)

Unifikation kann fehlschlagen...

```
[Int] -> [Int]   a -> [a]
        \        /
         \      /
         \/    \/
          (?,?)
```

Unifikation des...

- Argumenttyps erfordert ...a vom Typ `[Int]`
- Resultattyps erfordert ...a vom Typ `Int`
- Zusammen ...Unifikation schlägt fehl

Typüberprüfung von Ausdrücken

Hier... Funktionsapplikation

```
            f e
           /\   /\
          /       \
         /         \
    s -> t          u
    | /\           /\ \
    |  \          /   \
    |   \        /     |
    |   Unifikation    |
    |   von s und u    |
    \/                 \/
    s' -> t'           s'
        /\
         \
          \
      Resultattyp t'
```

Beobachtung:

- `s` und `u` müssen nicht übereinstimmen, sondern nur *unifizierbar* sein

Beispiel – `map` und `ord`

Betrachte...

```
map :: (a -> b) -> [a] -> [b]
ord :: Char -> Int
```

Unifikation von `a -> b` und `Char -> Int` liefert...

```
map :: (Char -> Int) -> [Char] -> [Int]
```

und somit weiter

```
map ord :: [Char] -> [Int]
```

Beispiel – `foldr`

Erinnerung...

```
foldr f s []     = s
foldr f s (x:xs) = f x (foldr f s xs)
```

Anwendungsbeispiel...

```
foldr (+) 0 [3,5,34] = 42
```

...suggeriert als allgemeinsten Typ

```
foldr :: (a -> a -> a) -> a -> [a] -> a
```


Beispiel – `foldr` 2(2)

Eine genauere überlegung zeigt...

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

Veranschaulichung:

```
                        b
                       /
                      /   a
                     \/  /
foldr f s []      = s  /      b
                       \/ ------------
foldr f s (x:xs)  = f x (foldr f s xs)
                     -----------------
                                b
```


Typüberprüfung bei polymorphen Funktionsdefinitionen

```
f :: t1 -> t2 -> ... -> tk -> t
f a1 a2 ... ak
  | b1   = e1
  | b2   = e2
  ...
  | bk   = ek
```

Drei Dinge sind zu überprüfen...

- Jeder Wächter `bi` muss vom Typ `Bool` sein
- Der Rückgabewert jeden Ausdrucks `ei` muss von einem Typ `si` sein, der mindestens so allgemein ist wie `t`, d.h. `t` muss Instanz von `si` sein
- Das Muster jeden Arguments `pi` muss mit dem Typ des formalen Parameters konsistent sein, also mit dem Typ `ti`.

Typüberprüfung und Typklassen

Betrachte...

```
member []     y = False
member (x:xs) y = (x==y) || member xs y
```

Die Verwendung von `(==)` im obigen Bsp. erzwingt...

```
member :: Eq a => [a] -> a -> Bool
```


Typüberprüfung und Überladen

Betrachte die Funktionsanwendung...

```
member e
```

mit

```
e      :: Ord b => [[b]]
```

Ohne Kontextinformation lieferte Unifikation...

```
member :: [[b]] -> [b] -> Bool
```

und somit weiter...

```
member e :: [b] -> Bool
```


Kontextanalyse

```
(Eq [b] , Ord b)
```

Analysieren und vereinfachen des Kontexts liefert...

- Kontextanforderungen beziehen sich auf Typvariablen
  `instance Eq a => Eq [a] where...` ...liefert somit `(Eq b, Ord b)`
- Wiederholen bis keine Instanz mehr anwendbar
- Vereinfachung des Kontexts mithilfe der durch `class` gegebenen Information liefert...
  `class Eq a => Ord a where...`
- Somit ...`Ord b`
- Insgesamt ...`member e :: Ord b => [b] -> Bool`

Zusammenfassung – Dreistufiger Prozess

Der dreistufige Prozess bestehend aus

- Unifikation
- Analyse (mit Instanzen)
- Simplifikation

...ist typisches Muster bei der kontextbehafteten Typanalyse in Haskell.

Typsysteme und Typinferenz

Informell...

- Typsysteme sind...
  – logische Systeme, die Aussagen der Form "exp hat den Typ tformalisieren und mit Hilfe von Axiomen und Regeln des Typsystems beweisen lassen
- Typinferenz bezeichnet...
  – den Vorgang mit Hilfe der Axiome und Regeln eines Typsystems den Typ eines Ausdrucks automatisch zu erschließen
  Stichwörter: Typinferenzalgorithmen, Unifikation

Typischer Ausschnitt aus einer Typgrammatik

...erzeugt die Sprache der Typen

$$\begin{array}{rcll} \tau & ::= & Int|Float|Char|Bool & \text{einfacher Typ} \\ & | & \alpha & \text{Typvariable} \\ & | & \tau \to \tau & \text{Funktionstyp} \\ \\ \sigma & ::= & \tau & \text{Typ} \\ & | & \forall \alpha.\ \sigma & \text{Typbindung} \end{array}$$

Sprechweisen:

- τ ...Typ
- σ ...Typschema

Typischer Ausschnitt aus einem Typ(inferenz)system 1(2)

...ordnet jedem (typisierbaren) Ausdruck der Sprache einen Typ der Typsprache zu

VAR $\dfrac{}{\Gamma \vdash var : \Gamma(var)}$

CON $\dfrac{}{\Gamma \vdash con : \Gamma(con)}$

COND $\dfrac{\Gamma \vdash exp : Bool \quad \Gamma \vdash exp_1 : \tau \quad \Gamma \vdash exp_2 : \tau}{\Gamma \vdash \text{ if } exp \text{ then } exp1 \text{ else } exp2 : \tau}$

APP $\dfrac{\Gamma \vdash exp : \tau' \to \tau \quad \Gamma \vdash exp' : \tau'}{\Gamma \vdash exp\ exp' : \tau}$

ABS $\dfrac{\Gamma[var \mapsto \tau'] \vdash exp : \tau}{\Gamma \vdash /x\text{->}exp : \tau' \to \tau}$

...

Typischer Ausschnitt aus einem Typinferenzsystem 2(2)

Typannahmen sind...

- partielle Funktionen, die Variablen auf Typschemata abbilden

Dabei ist $\Gamma[var_1 \mapsto \tau_1, \ldots, var_n \mapsto \tau_n]$ diejenige Funktion, die für var_i den Typ τ_i liefert und ansonsten wie Γ definiert ist.

Schematischer Unifikationsalgorithmus

$$\begin{array}{rcl} \mathcal{U}(\alpha,\alpha) & = & [] \\ \mathcal{U}(\alpha,\tau) & = & \begin{cases} [\tau/\alpha] & \text{falls } \alpha \notin \tau \\ error & \text{sonst} \end{cases} \\ \mathcal{U}(\tau,\alpha) & = & \mathcal{U}(\alpha,\tau) \\ \mathcal{U}(\tau_1 \to \tau_2, \tau_3 \to \tau_4) & = & \mathcal{U}(U\tau_2, U\tau_4)U \\ & & \quad \text{wobei } U = \mathcal{U}(\tau_1,\tau_3) \\ \mathcal{U}(\tau,\tau') & = & \begin{cases} [] & \text{falls } \tau = \tau' \\ error & \text{sonst} \end{cases} \end{array}$$

Hinweise:

- U ...(allgemeinster) Unifikator (i.w. eine Substitution)
- Gleichungsanwendung sequentiell von oben nach unten

Beispiel Unifikation/allgemeinste Unifikation

Betrachte... `a -> (Bool,c)` und `Int -> b`

- *Unifikation* ...Substitution `[Int/a,Float/c,(Bool,Float)/b]`
- *Allgemeinste Unifikation* ...Substitution `[Int/a,(Bool,c)/b]`

Beispiel zum Unifikationsalgorithmus

Aufgabe ...Unifikation der Typterme `a -> c` und `b -> Int -> a`

Lösung

$$\begin{array}{rcl} & & \mathcal{U}(\texttt{a -> c,b -> Int -> a}) \\ (\text{mit } U = \mathcal{U}(\texttt{a,b}) = [\texttt{b/a}]) & = & \mathcal{U}(U\texttt{c},U(\texttt{Int -> a}))U \\ & = & \mathcal{U}(\texttt{c, Int -> b})[b/a] \\ & = & [\texttt{Int -> b/c}][\texttt{b/a}] \\ & = & [\texttt{Int -> b/c,b/a}] \end{array}$$

Insgesamt ...`b -> Int -> b`

Essenz automatischer Typinferenzalgorithmen

...syntaxgesteuerte Anwendung der Regeln des Typinferenzsystems

Maßgeblich...

- Modifikation des Typinferenzsystems so, dass stets nur eine Regel anwendbar ist

Weiterführende und vertiefende Literatur 1(3)

Typen und Typsysteme, Typinferenz...

- Allgemein für funktionale Sprachen
  - Anthony J. Field, Peter G. Robinson. *Functional Programming*. Addison-Wesley, 1988 (Kapitel 7)
- Haskell-spezifisch
  - Simon Peyton Jones, J. Hughes. *Report on the Programming Language Haskell 98*. `http://www.haskell.org/report/`

Weiterführende und vertiefende Literatur 2(3)

- Übersicht
  - J. C. Mitchell. *Type Systems for Programming Languages*. In J. van Leeuwen (Hrsg.). *Handbook of Theoretical Computer Science, Vol. B: Formal Methods and Semantics*. Elsevier Science Publishers, 367-458, 1990.
- Grundlagen polymorpher Typsysteme
  - Robin Milner. *A Theory of Type Polymorphism in Programming*. Journal of Computer and System Sciences 17, 248-375, 1978.
  - L. Damas, Robin Milner. *Principal Type Schemes for Functional Programming Languages*. In Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages (POPL'82), 207-218, 1982

Weiterführende und vertiefende Literatur 3(3)

- Unifikationsalgorithmus
  - J. A. Robinson. *A Machine-Oriented Logic Based on the Resolution Principle*. Journal of the ACM 12(1), 23-42, 1965.
- Typsysteme und Typinferenz
  - Luca Cardelli. *Basic Polymorphic Type Checking*. Science of Computer Programming 8, 147-172, 1987.

Vorschau auf die nächsten Vorlesungstermine...

- Di, 01.06.2004, Pfingstferien, lehrveranstaltungsfrei
- Di, 15.06.2004, 16 - 18 Uhr, FH Hörsaal 2 (statt Di, 08.06.2004) (Achtung: anderer Ort)