Programmiersprachen

Alexandru Berlea

Institut für Informatik
TU München

Wintersemester 2006/2007

Parametrischer Polymorphismus

- Ein **parametrisierter Typ** definieren:

```
datatype 'a List = Nil | Cons of ('a * 'a List);
datatype 'a List = Cons of 'a * 'a List | Nil
```

  - 'a ist ein Bezeichner für einen bestimmten beliebigen Typ (Typ-Variable), der in Typ-Ausdrücken in Konstruktoren benutzt werden darf.

- Der Compiler erkennt den Parameter-Typ automatisch:

```
- Cons(1,Cons(2,Cons(3,Nil)));
val it = Cons (1,Cons (2,Cons 3)) : int List
- Cons(1.0,Cons(2.0,Cons(3.0,Nil)));
val it = Cons (1.0,Cons (2.0,Cons 3.0)) : real List
- Cons(true,Cons(true,Cons(false,Nil)));
val it = Cons (true,Cons (true,Cons false)) : bool List
```

Polymorphe Funktionen

```
datatype 'a List = Nil | Cons of ('a * 'a List)

fun length l =
    case l of
        Nil => 0
      | Cons(first, rest) => 1 + length rest;
val length = fn : 'a List -> int
```

Der Compiler leitet den allgemeinst möglichen Typ ab:
Hier ist 'a als *ein beliebiger Typ* zu lesen ⟹ `length` ist **polymorph**:

```
- length (Cons(1,Cons(2,Cons(3,Nil))));
val it = 3 : int
- length (Cons(1.0,Cons(2.0,Cons(3.0,Nil))));
val it = 3 : int
- length (Cons(true,Cons(true,Cons(false,Nil))));
val it = 3 : int
```

Der polymorphe Typ `list`

Ein polymorpher Typ `'a list` ist vordefiniert.
Konstruktoren:

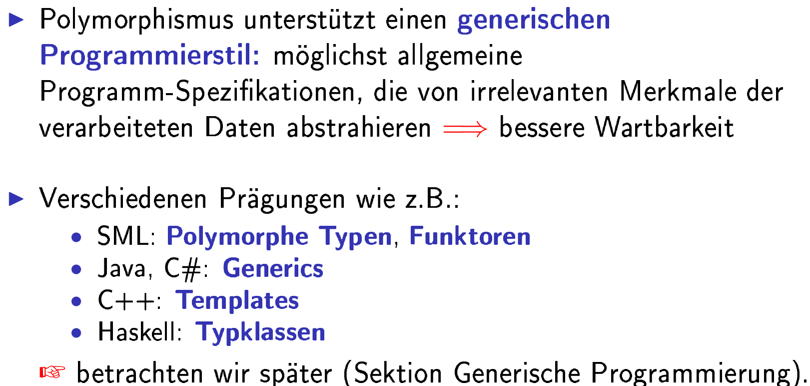
- nullstellig: `nil` (entspricht unserem `Nil`)
- einstellig: `::` (entspricht unseren `Cons`)
  - **infixiert**: `kopf::rest`
  - **rechtsassoziativ**: `1::(2::(3::nil))` ≡ `1::2::3::nil`
- Alternative Klammernotation:
  - `[]` ≡ `nil`
  - `[1,2,3]` ≡ `1::(2::(3::nil))` ≡ `1::[2,3]`

Der polymorphe Typ `list`

```
- fun length l = case l of
                     nil => 0
                   | first::rest => 1 + length rest;
val length = fn : 'a list -> int
- length [1,2,3];
val it = 3 : int
- length [true,true,false];
val it = 3 : int
```

- Alle Elemente einer Liste müssen vom selben Typ sein:
  [1,[1]] ist keine Liste!
- Liste von Listen von `ints`: `[[1,2,3],[4,5],[6],[7,8,9]]`

Generische Programmierung



Polymorphismus ist polymorph;-)

Formen von Polymorphismus:

- **Parametrischer Polymorphismus:**
  mit Typ-Variablen parametrisierte Typen (wie oben)

- **Subtyp-Polymorphismus:**
  - t_1 ist ein **Subtyp** von t_2, wenn er spezifischer ist (z.B. t_1 ist eine Unterklassse von t_2)
  - Eine Funktion, die Argumente von einem Typ annimmt, nimmt auch Argumente von jedem Subtyp an.

- **Ad-hoc-Polymorphismus:** Funktionen mit (beliebig) verschiedenen Typen haben den selben Namen (**Überladen/overloading**)
  - z.B. arithmetische Funktionen
  - nur endlich viele Typen können angenommen werden

Überblick

1. **Typ-Inferenz**
   - Typ-Inferenz vs. Typ-Checking
   - Inferenz-Regeln
   - Lösung
   - Unifikation
   - Polymorphe Funktionen
   - Typ-Annotationen
   - Schlussbemerkungen

Typ-Inferenz

Zum Type-Checking muss man den Typ für jeden Ausdruck im Programm kennen. Ansätze:

- Klassisches Type-Checking:
  - **Typ-Deklarationen** für Variablen/Funktionen **erforderlich**.
  - Typherleitung aus den Typen der Teilausdrücke (bottom-up), z.B.

$$e_1 : int \wedge e_2 : int \Rightarrow e_1 + e_2 : int$$

Typ-Inferenz

Zum Type-Checking muss man den Typ für jeden Ausdruck im Programm kennen. Ansätze:

- **Klassisches Type-Checking:**
  - **Typ-Deklarationen** für Variablen/Funktionen **erforderlich**.
  - Typherleitung aus den Typen der Teilausdrücke (bottom-up), z.B.

$$e_1 : int \wedge e_2 : int \Rightarrow e_1 + e_2 : int$$

- **Typ-Inferenz:**
  - **Typ-Deklarationen nicht nötig**
  - Typherleitung (insb. für Variablen/Funktionen) aus dem Kontext
    - der Typ eines Ausdrucks wird sowohl durch den Typ der Bestandteile (bottom-up) als auch durch den Typ der umgebenden Ausdrücke (top-down) eingeschränkt, z.B.

$$e_1 : int \wedge e_2 : int \Leftrightarrow e_1 + e_2 : int$$

Terminologie

- Sowohl beim klassischen Type-Checking als auch bei der Typ-Inferenz werden Typen hergeleitet/**inferiert**:
- Unterschied:
  - Klassisches Type-Checking:
    - nur Typen der Zwischenergebnisse werden **inferiert**

```
1 + o.a().getInt();
```

      Die Typen von `o.a()` und `o.a().getInt()` werden inferiert.
    - Variablen- u. Fkt.-Typen müssen vorgegeben werden

```
int fact(int n) {if (n<=0) return 1
                 else return n * fact(n-1);}
```

  - Typ-Inferenz: (möglichst) alle Typen werden **inferiert**

```
fun fact n = if n <= 0 then 1 else n*fact(n-1)
```

SML Inferenz-Regeln

- Typen werden mit Hilfe von **Inferenz-Regeln** berechnet.

  Angabe der Beziehung zwischen dem Typ eines Ausdruckes und den Typen seiner Teilausdrücke.

- **Bsp.:** Wenn x und y vom Typ `int` sind, dann ist $x + y$ auch vom Typ `int` (und umgekehrt). Schreibweise:

$$\frac{x : \text{int} \qquad y : \text{int}}{x{+}y : \text{int}}$$

Inferenz-Regeln

Inferenz-Regel: zeigen, welche Typen *gleich* sein (*unifizieren*) müssen.

Case:
$$\frac{e : t_1 \quad p_1 : t_1\texttt{,}\ \ldots\texttt{,}\ p_n : t_1 \quad e_1 : t_2\texttt{,}\ \ldots\texttt{,}\ e_n : t_2}{\texttt{case}\ e\ \texttt{of}\ p_1\ \texttt{=>}\ e_1\ \texttt{|}\ \ldots\texttt{|}\ p_n\ \texttt{=>}\ e_n\ :\ t_2}$$

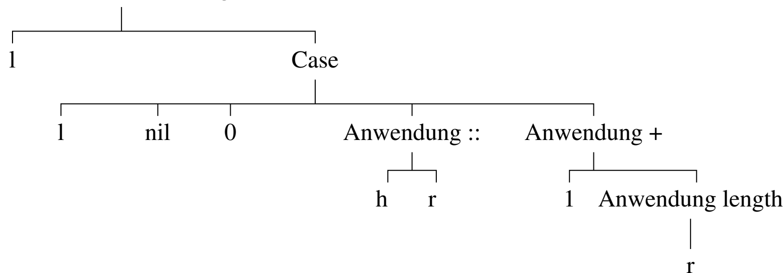
If:
$$\frac{p : \texttt{bool} \quad A : t_1 \quad B : t_1}{\texttt{if}\ p\ \texttt{then}\ A\ \texttt{else}\ B : t_1}$$

Funktionsanwendung/Konstruktor
$$\frac{f : t_1 \mapsto t_2 \quad a : t_1}{f\ a : t_2}$$

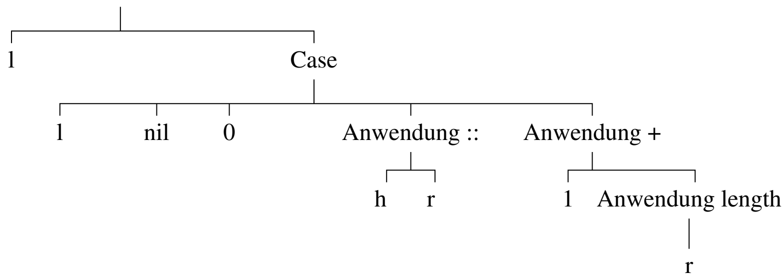
Funktionsdefinition:
$$\frac{x : t_1 \quad e : t_2}{\texttt{fun}\ name\ x\ \texttt{=}\ e : t_1 \mapsto t_2}$$

Typ-Inferenz: Bsp.

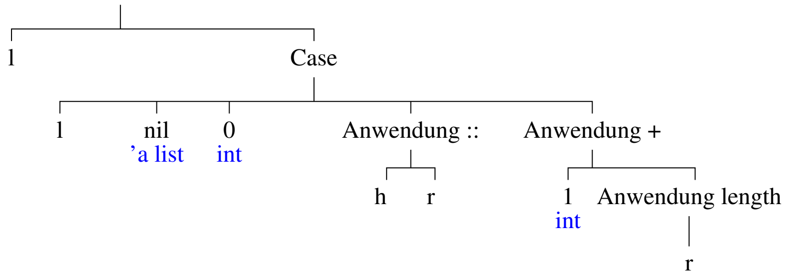
```
fun length l = case l of
                    nil => 0
                  | h::r => 1 + length r
```



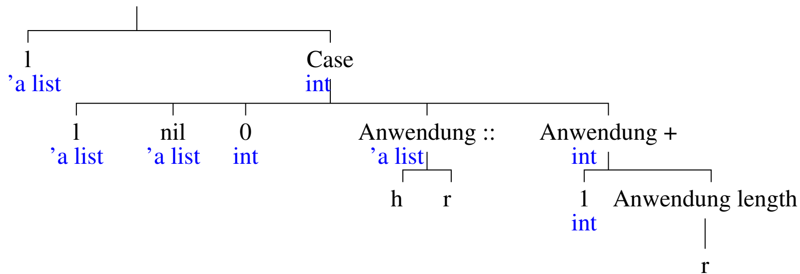
Typ-Inferenz: Bsp.



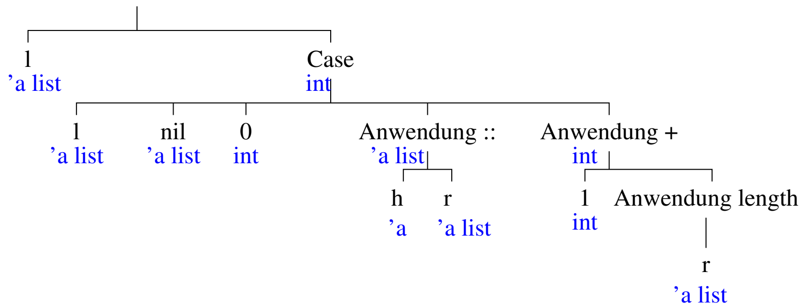
Typ-Inferenz: 0-stellige Konstrukt./Konstanten



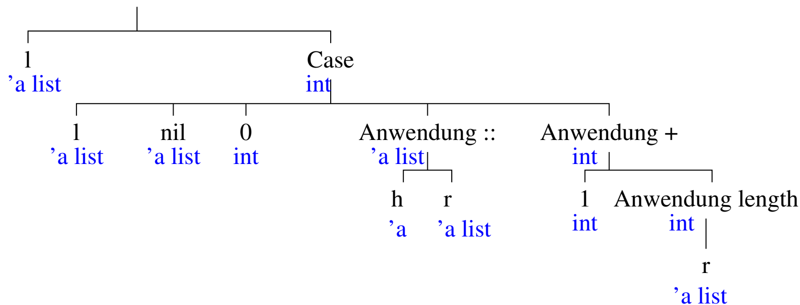
Typ-Inferenz: Case



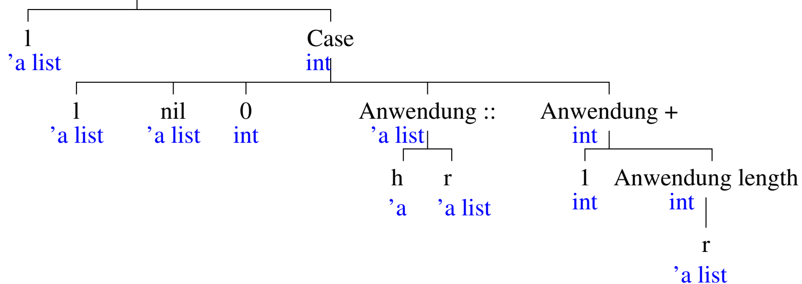
Typ-Inferenz: Konstruktor ::



Typ-Inferenz: Anwendung length



Typ-Inferenz: Fkt.-Definition



Typ-Inferenz: Idee

Allgemeine Lösung:

- Weise jedem Teilausdruck einen unbekannten Typ (eine Typ-Variable) zu.
- Wende die entsprechenden Inferenz-Regel an jedem Teilausdruck $\Longrightarrow$ System von Term-Gleichungen
- Löse das Term-Gleichungssystem
  - **lösbar**: Lösung gibt Typen für jeden Teilausdruck (inkl. Variablen und Funktionen) an
  - **unlösbar**: Typ-Fehler

Unifikation

- Lösen von Systemen von Term-Gleichungen = **Unifikation**
  - Lösung = **Substitution** (Unifikator) der Variablen, die die Terme **strukturell gleich** machen.
  - Bsp.:
    - `('a -> ('a * 'b)) list = (int -> ('c * 'c) list)`
    - Lösung: `{'a` $\mapsto$ `int, 'b` $\mapsto$ `int, 'c` $\mapsto$ `int}`

- Die Lösung ist nicht immer eindeutig
  - Bsp.:
    - `('a -> 'a) = ('b -> 'c)`
    - Lösungen: `{'a` $\mapsto$ `int, 'b` $\mapsto$ `int, 'c` $\mapsto$ `int}`
      `{'a` $\mapsto$ `bool, 'b` $\mapsto$ `bool, 'c` $\mapsto$ `bool}`
      ...

Unifikation

- Satz: Ein System von Term-Gleichungen hat entweder:
  - **keine** Lösung
  - **eine eindeutige** Lösung
  - **beliebig viele** Lösungen. In diesem Fall gibt es eine allgemeinste Lösung.

Unifikation

- Satz: Ein System von Term-Gleichungen hat entweder:
  - **keine** Lösung
  - **eine eindeutige** Lösung
  - **beliebig viele** Lösungen. In diesem Fall gibt es eine allgemeinste Lösung.

- Die allgemeinste Lösung = **Most general unifier** (mgu)
  - σ=mgu falls für jeden anderen Unifikator θ gilt:

    $$\theta = \theta' \circ \sigma$$

    für eine Substitution θ'.
  - Bsp.:
    - `('a -> 'a)= ('b -> 'c)`
    - mgu $= \{\texttt{'b} \mapsto \texttt{'a}, \texttt{'c} \mapsto \texttt{'a}\}$
    - $\{\texttt{'a} \mapsto \texttt{int}, \texttt{'b} \mapsto \texttt{int}, \texttt{'c} \mapsto \texttt{int}\} = \{\texttt{'a} \mapsto \texttt{int}\} \circ \texttt{mgu}$

Berechnung des mgu

- **Eingabe:** Zwei Terme T_1 und T_2
- **Ausgabe:** failure, wenn T_1 und T_2 nicht unifizierbar sind;
  ihr mgu, sonst.
- Idee:
  - schreibe Term-Gleichungen in Gleichungen zwischen entsprechenden Teil-Termen um;
  - nutze einen Keller als Workset, um noch nicht gelöste Gleichungen zu speichern;
  - nutzte eine Liste von Substitutionen θ, die die Ausgabe aufsammelt.

Algorithmus zur Berechnung des mgu

θ := $\emptyset$; push(stack, $T_1 = T_2$); failure = false;
while not empty(stack) and not failure do
  hole X = Y vom stack runter
  case
    X ist eine Variable, die in Y nicht auftritt:
      ersetze X durch Y im stack und in θ
      füge X $\mapsto$ Y zu θ hinzu
    Y ist eine Variable, die in X nicht auftritt:
      ersetze Y durch X im stack und in θ
      füge Y $\mapsto$ X zu θ hinzu
    X und Y sind identische Konstanten oder Variablen: continue
    X ist $f(X_1, \ldots, X_n)$ und Y ist $f(Y_1, \ldots, Y_n)$ mit f=Operator:
      push(stack, $X_1 = Y_1$, $X_2 = Y_2$, ..., $X_n = Y_n$)
    sonst:
      failure := true
if failure then output *failure* else output θ

mgu-Berechnung: Beispiel

- Zu unifizieren: T_1=`(('a * 'b) list) * ('c -> 'd)` und
  T_2=`('c list) * ('a * 'd -> 'b)`
- Stack $s \equiv [T_1 = T_2]$; Substitution $\theta \equiv \{\}$
- $s \equiv$ [`('a * 'b) list = 'c list`; `'c -> 'd = 'a * 'd -> 'b`];
  $\theta \equiv \{\}$
- $s \equiv$ [`'a * 'b = 'c`; `'c -> 'd = 'a * 'd -> 'b`];$\theta \equiv \{\}$
- $s \equiv$ [`'a * 'b -> 'd = 'a * 'd -> 'b`]; $\theta \equiv \{$'c$\mapsto$ 'a * 'b$\}$
- $s \equiv$ [`'a * 'b = 'a * 'd`;`'d = 'b`]; $\theta \equiv \{$'c$\mapsto$ 'a * 'b$\}$
- $s \equiv$ [`'a = 'a`;`'b = 'd`;`'d = 'b`]; $\theta \equiv \{$'c$\mapsto$ 'a * 'b$\}$
- $s \equiv$ [`'b = 'd`;`'d = 'b`]; $\theta \equiv \{$'c$\mapsto$ 'a * 'b$\}$
- $s \equiv$ [`'d = 'd`]; $\theta \equiv \{$'c$\mapsto$ 'a * 'd, 'b$\mapsto$'d$\}$
- $s \equiv$ []; $\theta \equiv \{$'c$\mapsto$ 'a * 'd, 'b$\mapsto$'d$\}$

Der *occurs check* Test

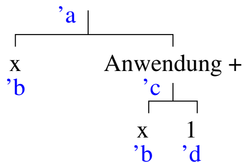
```
θ := ∅; push(stack, T1 = T2); failure = false;
while not empty(stack) and not failure do
  hole X = Y vom stack runter
  case
    ....
    X ist eine Variable, die in Y nicht auftritt:
      ersetze Y durch X im stack und in θ
      füge X = Y zu θ hinzu
    ....

if failure then output failure else output θ
```

- ...stellt sicher, dass die Unifikation terminiert; Z.B. gibt es keine endliche gemeinsame Instanz von `'a` und `'a list`;

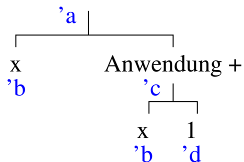
Typ-Inferenz: Beispiel

```
fun inc x = x + 1
```



Typ-Inferenz: Beispiel

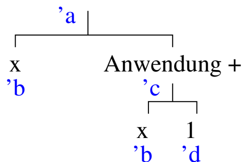
```
fun inc x = x + 1
```



$$\left\{ \begin{array}{l} \texttt{'a = 'b -> 'c} \quad \textbf{(Fkt.-Def. inc)} \\ \\ \\ \\ \end{array} \right.$$

Typ-Inferenz: Beispiel

```
fun inc x = x + 1
```

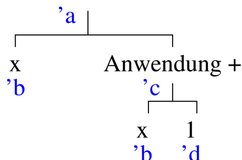


'a = 'b -> 'c (Fkt.-Def. inc)

'c = int (Anwendung +)
'b = int
'd = int

Typ-Inferenz: Beispiel

```
fun inc x = x + 1
```



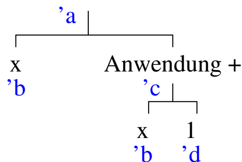
'a = 'b -> 'c (Fkt.-Def. inc)

'c = int (Anwendung +)
'b = int
'd = int

'd = int (Konstante 1)

Typ-Inferenz: Beispiel

```
fun inc x = x + 1
```



$$\left\{\begin{array}{ll} 'a = 'b \text{ -> } 'c & \text{(Fkt.-Def. inc)} \\ 'c = \text{int} & \text{(Anwendung +)} \\ 'b = \text{int} & \\ 'd = \text{int} & \\ 'd = \text{int} & \text{(Konstante 1)} \end{array}\right.$$

$$\Longrightarrow \left\{\begin{array}{l} 'a = \text{int -> int} \\ 'b = \text{int} \\ 'c = \text{int} \\ 'd = \text{int} \end{array}\right.$$

Typ-Inferenz: Rekursive Funktionen

- Der Typ-Inferenz-Algorithmus handelt korrekt **rekursive Funktionen**
- Bsp.:

```
fun sum n = if n<=0 then 0
            else n + sum (n-1)
```

- Der korrekte Typ erhält man durch Benutzung der selben Typ-Variablen in Fkt.-Def und rek. Anwendung ☞ Übung

```
val sum = fn : int -> int
```

Typ-Inferenz: Polymorphe Funktionen

- **Polymorphe Funktionen** sind Fkt., deren inferierten Funktionstyp Typ-Variablen enthält.

```
fun id x = x
val id = fn : 'a -> 'a
```

- Problem: wie typt man `(id 1,id true)`?
  - `'a` ist frei in `'a ->'a`
  - `'a` muss bei jeder Anwendung mit dem Operanden-Typ unifizieren:

    $\{'a = \text{int}, 'a = \text{bool} \Longrightarrow$ Typ-Fehler??

- Lösung:
  - univ. Quantifizierung: $\forall$`'a.'a->'a` **(Typ-Schema)**
  - binde frische Typ-Variablen bei jeder (nicht-rek.) Anwendung

    $\{'a_1 = \text{int}, 'a_2 = \text{bool} \Longrightarrow$`(id 1,id true):int*bool`

Typ-Inferenz: Benutzer-definierte Konstruktoren

Wir definieren:

```
datatype 'a T = F of 'a T -> 'a
fun f (F x) = F (x (F x))
```

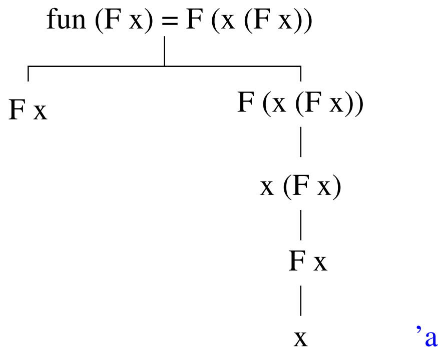
Welchen Typ hat `f`?

Typ-Inferenz: Benutzer-definierte Konstruktoren

Wir definieren:

```
datatype 'a T = F of 'a T -> 'a
fun f (F x) = F (x (F x))
```

Welchen Typ hat `f`?

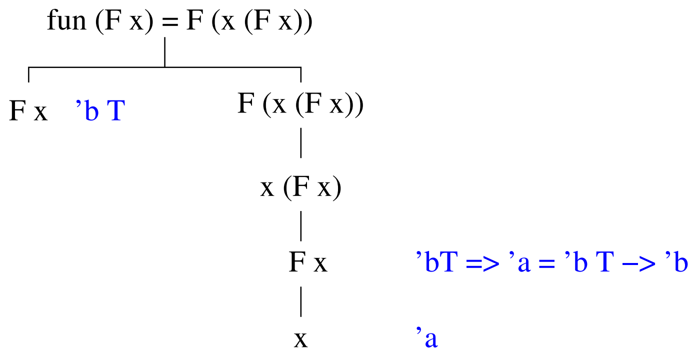


Typ-Inferenz: Benutzer-definierte Konstruktoren

Wir definieren:

```
datatype 'a T = F of 'a T -> 'a
fun f (F x) = F (x (F x))
```

Welchen Typ hat `f`?

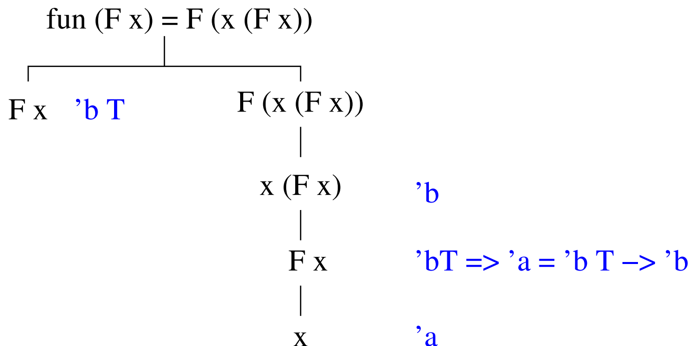


Typ-Inferenz: Benutzer-definierte Konstruktoren

Wir definieren:

```
datatype 'a T = F of 'a T -> 'a
fun f (F x) = F (x (F x))
```

Welchen Typ hat `f`?

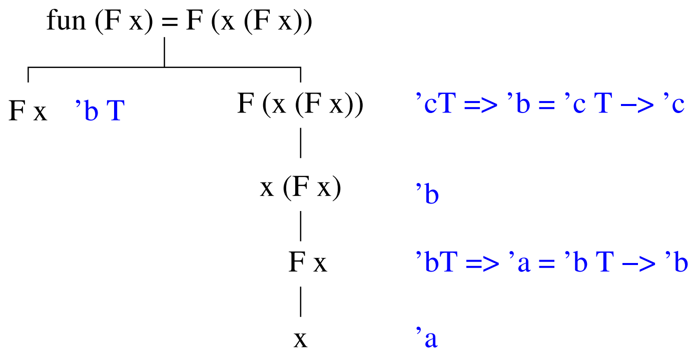


Typ-Inferenz: Benutzer-definierte Konstruktoren

Wir definieren:

```
datatype 'a T = F of 'a T -> 'a
fun f (F x) = F (x (F x))
```

Welchen Typ hat `f`?

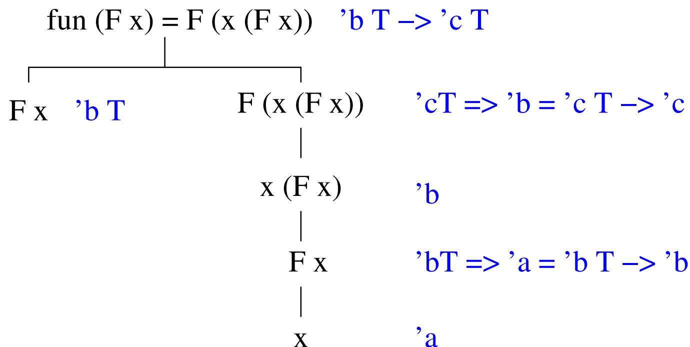


Typ-Inferenz: Benutzer-definierte Konstruktoren

Wir definieren:

```
datatype 'a T = F of 'a T -> 'a
fun f (F x) = F (x (F x))
```

Welchen Typ hat `f`?



Typ-Inferenz: Fehlermeldungen

- Hat das Gleichungssystem keine Lösung $\Longrightarrow$ Typ-Fehler
- erwünscht: welche Stelle im Programm/Ausdruck den Fehler verursacht
- Idee: Anordnung und Lösen der Gleichungen durch einen DFS-Durchlauf leiten (**Syntax-gerichtet**)
  [☞ Compilerbau Vorlesung]

Typ-Annotationen

- Wenn dem Programmierer der hergeleitete Typ eines Ausdrucks zu allgemein ist, kann er ihn mit Hilfe von **Typ-Annotationen** einschränken.

```
[];
val it = [] : 'a list
[] : int list;
val it = [] : int list

fun f x = [x];
val f = fn : 'a -> 'a list
fun f x = [x] :int list;
val f = fn : int -> int list
fun f (x :int) = [x];
val f = fn : int -> int list
```

- Die Typ-Annotationen werden als zusätzliche Constraints im Gleichungssystem zur Typ-Inferenz aufgenommen.

Typ-Annotationen

- Der angegebene Typ muss eine Instanz des hergeleiteten Typs sein:

```
[1] : 'a list;
stdIn:17.1-17.17 Error: expression doesn't match constraint
  expression: int list
  constraint: 'a list
in expression: 1 :: nil: 'a list

fun f x = (x,x) : 'a * 'b;
stdIn:2.6-2.20 Error: expression doesn't match constraint
    expression: 'a * 'a
    constraint: 'a * 'b
in expression: (x,x): 'a * 'b
```

Typ-Inferenz: Schlussbemerkungen

- Typen und Typ-Systemen erlauben automatisches Typ-Checking $\Longrightarrow$ Vermeiden von Laufzeitfehlern
- Typ-Inferenz entlastet den Programmierer und unterstützt Generizität
- Ähnliche Methoden können in Übersetzerbau und Software-Engineering eingesetzt werden, um Programm-Eigenschaften herauszufinden. (☞ Programmoptimierung).

Überblick

2. **Variablen**
Variablendefinitionen
Variablengültigkeit
Variablensichtbarkeit

Variablendefinitionen

Eine Variable v ist ein Paar der Form $(name, wert)$ (geschrieben auch: $name \leftarrow wert$).

- ... bestehen aus *name* und einen Ausdruck für *wert*
- heißen auch **Variablen-Bindungen** (*variable bindings*)
- können explizit oder implizit sein

Variablendefinitionen

- **Explizite Definition**:

```
val x = 1*2;
```

- **Implizite Definition**:
  - via Funktionsaufrufe

    ```
    fun f x = 42
    f (25*4)
    ```

    Der Aufruf `f (25*4)` bindet `x` zu dem Wert von `25*4`.
  - via Pattern-Matching mit Variablen-Bindungen

Variablengültigkeit

- Die **Gültigkeit** einer Variable (*scope*) ist die Menge der Programmpunkte, an denen ihre Definition gilt.
- Top-level Variablen (definiert mit `val name = expr` in der Interpreter-Umgebung) sind gültig an allen nachfolgenden Programmpunkten:

```
val x = 1*2;
val y = x+1;
```

Variablengültigkeit

- Parameter-Variablen (Funktionsargumente) sind gültig im Rumpf der Funktion

```
fun f x = x+1
```

- Pattern-Variablen sind gültig in der entsprechenden rechten Seite.

```
val description = case f of
                        Rot => "pure red"
                      | Blau => "pure blue"
                      | RGB(x,y,_) =>
                        if (x=y) then "kind of yellow"
                        else "something else";
```

Benutzerdefinierte Gültigkeitsbereiche

... können mit Hilfe des `let`-Ausdrucks eingeführt werden:

```
let
 val name = ausdruck
in
 ausdruck'
end
```

- Der Scope der Variable *name* ist *ausdruck'*.
- Der Wert des let-Ausdrucks ist der Wert von *ausdruck'*.

```
let
   val x = 1 + 1
in
 10 * x
end
val it = 20 : int
```

Der let-Ausdruck

... kann auch den Scope einer Funktionsdefinition einschränken

```
let
  fun square x = x*x
in
  square 2
end;
val it = 4 : int
- square 3;
stdIn:75.1-75.7 Error: unbound variable or constructor: square
```

Geschachtelte let-Ausdrücke

Oft möchte man geschachtelte Gültigkeitsbereiche:

```
let val x = 1
in let val y = x+1
   in x+y
   end
end;
val it = 3 : int
```

Äquivalent kann man schreiben:

```
let
  val x = 1
  val y = x+1
in x+y
end;
val it = 3 : int
```

Der let-Ausdruck

Im Allgemeinen:

```
let
 val name₁ = ausdruck₁
 val name₂ = ausdruck₂
  ........................
 val nameₙ = ausdruckₙ
in
 ausdruck
end
```

- Der Scope der Variable $name_i$ besteht aus $ausdruck_{i+1}$, ..., $ausdruck_n$ und *ausdruck*.
- Der Wert des let-Ausdrucks ist der Wert von *ausdruck*.

Der let-Ausdruck: Beispiel

```
let
  val increment = 2
  fun add x = x + increment
in
  add 4
end;
val it = 6 : int
```

Statischer Gültigkeitsbereich

- Der Gültigkeitsbereich einer Variablendefinition in SML und in den meisten modernen Programmiersprachen ist durch die (**statische**) Struktur des Programmtextes definiert.
  $\Longrightarrow$ *static scoping*

- D.h., welche Variablen-Definitionen an einem Programmpunkt gültig sind, hängt nur von der (statischen) Struktur des Programmtextes ab.
  $\Longrightarrow$ *static/lexical scoping* $\equiv$ *static/lexical variable binding*

Statischer Gültigkeitsbereich: Bsp.

Variablen-Definitionen gültig am Programmpunkt •

```
let
  val sep = ";"
  fun set2String l =
    let fun doit l1 =
          case l1 of
           nil => ""
          |x::nil => Int.toString x
          |x::rest => • (Int.toString x)^sep^(doit rest)
    in "{"^(doit l)^"}" end
in set2String [1,2,3] end
val it = "{1;2;3}" : string
```

Statischer Gültigkeitsbereich: Bsp.

Variablen-Definitionen gültig am Programmpunkt •

```
let
  val sep = ";"
  fun set2String l =
    let fun doit l1 =
          case l1 of
           nil => ""
          |x::nil => Int.toString x
          |x::rest => (Int.toString x)^sep^(doit rest)
    in • "{"^(doit l)^"}" end
in set2String [1,2,3] end
val it = "{1;2;3}" : string
```

Statischer Gültigkeitsbereich: Bsp.

Variablen-Definitionen gültig am Programmpunkt •

```
let
  val sep = ";"
  fun set2String l =
    let fun doit l1 =
          case l1 of
           nil => ""
          |x::nil => Int.toString x
          |x::rest =>  (Int.toString x)^sep^(doit rest)
    in  "{"^(doit l)^"}" end
in • set2String [1,2,3] end
val it = "{1;2;3}" : string
```

Dynamischer vs. statischer Scoping

- **Dynamic scoping:** welche Variablen-Definition an einem Programmpunkt gültig sind, hängt davon ab, wie dieser bei der Laufzeit erreicht wird.

```
int i = 1;

int f(){•return i;}

int g{
  int i = 2;
  return f();
}
```

  - unter **statischem Scoping**: g() liefert 1
  - unter dynamischen Scoping angenommen: g() liefert 2
    $\Longrightarrow$ Die referentielle Transparenz ist verletzt

Dynamischer Gültigkeitsbereich: Bsp.

- Beispiel (Scheme):

```
(define mult (lambda (x y) (* x y)))
(define fact (lambda (n)
                (if (= n 0)
                    1
                    (mult (fact (- n 1) n)))))
(fact 3)
6

(define mult (lambda (x y) (y)))
(fact 3)
3
```

- Grund: Die Bindung der top-level Variablen in Scheme ist dynamisch.

Variablensichtbarkeit

Eine Variablen-Definition ist an einem Programmpunkt P zu einem bestimmten Zeitpunkt t **sichtbar**, wenn:

1. die Variablen-Definition an P zum Zeitpunkt t gültig ist, und
2. alle anderen gültigen Variablen-Definitionen mit dem selben Namen zu einem früheren Zeitpunkt stattgefunden haben.

Variablen-Sichtbarkeit: Beispiel

```
let
  val x = 1
in
  (let
     val x = 2
   in
     ● x+1
   end) + ● x
end
```

- ● gültig: $x \leftarrow 1$, $x \leftarrow 2$
  sichtbar: $x \leftarrow 2$
- ● gültig: $x \leftarrow 1$
  sichtbar: $x \leftarrow 1$

Variablen-Sichtbarkeit: Beispiel

```
fun fact n = if n=1 then 1 else n*(fact(n-1))
```

Auswertung `fact(3)`
    `n` ← 3
    Auswertung `fact(2)`
        `n` ← 2
        Auswertung `fact(1)`
            `n` ← 1
            Rückgabe 1
        Auswertung n`*fact(1)`
        Rückgabe 2
    Auswertung n`*fact(2)`
    Rückgabe 6

Zeit	Var-Def (implizit)	gültig	sichtbar
t_1:	`(fact 3)`	$n \leftarrow 3$	$n \leftarrow 3$
t_2:	`(fact 2)`	$n \leftarrow 3$, $n \leftarrow 2$	$n \leftarrow 2$
t_3:	`(fact 1)`	$n \leftarrow 3$, $n \leftarrow 2$, $n \leftarrow 1$	$n \leftarrow 1$

- Der **Kontext** eines Programmpunkts P zu einem bestimmten Zeitpunkt t = Menge sichtbarer Variablen-Definitionen am P zum Zeitpunkt t.
  $\Longrightarrow$ ist ein dynamischer Konzept: Dem selben Programmpunkt können bei der Laufzeit verschiedene Kontexte zu verschiedenen Zeitpunkten entsprechen.

$$Kontext(P)_t = \{n \leftarrow 1\}$$
$$Kontext(P)_{t+\Delta t} = \{n \leftarrow 2\}$$

Kontext: Beispiel

```
fun fact n = • if n=1 then 1
               else n * (fact (n-1))
```

Kontext von •:

Zeit	Kontext
t_1: (fact 3)	$n \leftarrow 3$
t_2: (fact 2)	$n \leftarrow 2$
t_3: (fact 2)	$n \leftarrow 1$