Programmierung und Modellierung

Typüberprüfung und Typinferenz

Martin Wirsing

in Zusammenarbeit mit
Moritz Hammer

Inhalt

- Kap. 5 Typüberprüfung und Typinferenz
  1. Typaussagen
  2. Typisierungsaxiome und –regeln
  3. Typinferenz und Unifikation

Wozu Typüberprüfung?

Robin Milner: "*Well-typed programs do not go wrong*".

- **Dynamischer Typfehler:** Die Auswertung scheitert wegen falscher Operanden. Beispiele: `true + 8` oder `5(3)`(Funktionsanwendung).
- Die **statische Typüberprüfung** weist solche Programme bereits vor deren Ausführung zurück.
  - Allerdings werden auch manche "gute Programme" zurückgewiesen:

    ```
    if true then 1 else 5(3)
    ```

    "Irgendwie" stimmt aber doch etwas nicht mit diesem Programm!
  - Statische Typüberprüfung definiert eine intuitiv begreifbare Teilmenge derjenigen Programme, die keine dynamischen Typfehler exhibieren.

ML Folklore: "*If it typechecks, it works*".

5.1 Typaussagen

- Eine **Typaussage** (auch **Typisierungsurteil**, engl.: *typing judgment*) ist eine Aussage der Form

  $\Gamma \triangleright$ `e : typ`

  wobei
  - e ein SML-Ausdruck ist,
  - `typ` ein SML-Typ ist und
  - Γ eine **Menge von (Typ-)Bindungen** (spezielle **Typ-Constraints**) der Form x_i`:typ`$_i$ ist, welche den Bezeichnern (Namen) in e Typen zuweist. Dabei darf kein Bezeichner in Γ zwei verschiedene Typen zugewiesen bekommen!
- Solch eine Menge von Typbindungen heißt **Typkontext** oder **Typzuweisung** (engl.: **type assignment**, auch **typing context**).
- **Bedeutung einer Typaussage**: "**Unter der Annahme, dass die Bezeichner die in Γ angegebenen Typen haben, hat e den Typ `typ`.**"
- **Beispiel:**

```
a:int ▸ fn x => a + 2*x : int -> int
```

5.2 Typisierungsaxiome und Typisierungsregeln

- Typisierungsaussagen werden **formal hergeleitet** aus
  - **Typaxiomen** mithilfe von
  - **Typisierungsregeln** (engl.: **typing rules**).
- **Typaxiome** sind elementare Typaussagen wie etwa
  - `{} ▸ 7 : int` oder
  - `{} ▸ 1.03 : real` .
- Eine **Typisierungsregel** hat die Form

  $P_1, \ldots, P_m \vdash K$, auch geschrieben $\dfrac{P_1, \ldots, P_m}{K}$

  wobei die P_i und K Typaussagen sind. Die P_i heißen **Prämissen**, K heißt **Konklusion** der Regel.

  Bedeutung : Gelten alle P_i , so soll auch K gelten.

Herleitung von Typaussagen

- Eine **Herleitung** einer Typaussage A ist eine Folge $A_1, \ldots, A_n$ von Typaussagen derart, dass
  - $A_n = A$ und
  - jedes A_i entweder ein Typaxiom ist, oder
  - Konklusion einer Typisierungsregel,
    deren Prämissen unter den $A_1, \ldots, A_{i-1}$ sind.

- Ein **Herleitungskalkül** besteht aus einer Menge von Axiomen und Regeln

  Beispiele
  - Herleitungskalkül für Typaussagen
  - Herleitungskalkül für logische Aussagen
  - Herleitungskalkül für Prädikatenlogik

5.2.1 Typisierungsregel für Kartesisches Produkt

- **Typisierungsregel [x] für Kartesisches Produkt**

  $\Gamma_1 \triangleright e_1 : typ_1$

  $\Gamma_2 \triangleright e_2 : typ_2$

  $\vdash \Gamma_1 \cup \Gamma_2 \triangleright (e_1, e_2) : typ_1 * typ_2$

  wobei Γ_1 und Γ_2 **kompatibel** sein müssen.

- **Definition**

  Γ_1 und Γ_2 sind **kompatibel**, wenn $\Gamma_1 \cup \Gamma_2$ eine Typzuweisung ist, d.h. wenn aus $x : typ_1 \in \Gamma_1$ und $x : typ_2 \in \Gamma_2$ folgt $typ_1 = typ_2$.

- **Beispiel**

  $\{\} \triangleright 17*5 : int$

  $\{\} \triangleright true : bool$

  $\vdash \{\} \triangleright (17*5, true) : int * bool$

5.2.1 Typisierungsregel für Kartesisches Produkt

- **Bemerkung**

  Die Typisierungsregel für das kartesische Produkt ist eigentlich ein **Regelschema**, das für unendlich viele einzelne Typisierungsregeln steht:

  je eine für jedes konkret gegebene Γ_1, Γ_2, `e1, e2,` $\texttt{typ}_1, \texttt{typ}_2$, so dass die Kompatibilitätsbedingung erfüllt ist.

5.2.2 Typisierungsregeln für Basistypen

- **Typisierungsaxiom [0] für die Konstante 0**

  $\{\} \triangleright \texttt{0 : int}$

- **Typisierungsregel [op] für arithmetische Operationen**

  $\Gamma_1 \triangleright e_1 : \texttt{int}$

  $\Gamma_2 \triangleright e_2 : \texttt{int}$

  $\vdash \Gamma_1 \cup \Gamma_2 \triangleright e_1 \texttt{ op } e_2 : \texttt{int}$

  wobei $\texttt{op} \in \{\texttt{+}, \texttt{*}, \texttt{/}, \texttt{div}, \texttt{mod}\}$ und Γ_1, Γ_2 kompatibel.

- Analoge Regeln gibt es für `real`, `string`, `char` und `bool`.
- In Zukunft erwähnen wir die Kompatibilitätsbedingung nicht mehr explizit.
- **Beispiel**

  $\{\} \triangleright \texttt{17 : int}$

  $\{\} \triangleright \texttt{ 5 : int}$

  $\vdash \{\} \triangleright \texttt{17*5 : int}$

5.2.3 Typisierungsregel für Fallunterscheidung

- **Typisierungsregel [IF] für Fallunterscheidung**

$$\frac{\Gamma_1 \triangleright e_1 : \texttt{bool} \quad \Gamma_2 \triangleright e_2 : \texttt{typ} \quad \Gamma_3 \triangleright e_3 : \texttt{typ}}{\vdash \Gamma_1 \cup \Gamma_2 \cup \Gamma_3 \triangleright \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \texttt{typ}}$$

- **Achtung**

  Beide Zweige der Fallunterscheidung müssen denselben Typ haben.

- **Beispiel**

$$\frac{\{\} \triangleright x > 0 : \texttt{bool} \quad \{\} \triangleright x-1 : \texttt{int} \quad \{\} \triangleright 0 : \texttt{int}}{\vdash \{\} \triangleright \texttt{if } x > 0 \texttt{ then } x-1 \texttt{ else } 0 : \texttt{int}}$$

5.2.4 Typisierungsaxiom für Bezeichner

- **Typisierungsaxiom [ID] für Bezeichner**

  `{x:typ} ⊢ x : typ`

  wobei `x` ein Bezeichner und `typ` ein Typ ist.

- **Beispiel**

  `{x:int} ⊢ x : int`

  `{} ⊢ 5 : int`

  `⊢ {x:int} ⊢ x*5 : int`

5.2.5 Typisierungsregel für Funktionen

- **Typisierungsregel [FN] für Funktionen**

$$\frac{\Gamma \cup \{x{:}typ_0\} \triangleright e : typ_1}{\Gamma \vdash \texttt{fn } x \texttt{ =>} e : typ_0 \texttt{ -> } typ_1}$$

- **Beispiel**

$$\dfrac{\dfrac{\{\} \triangleright 2 : \texttt{int}\ [2] \qquad \{x{:}\texttt{int}\} \triangleright x : \texttt{int}\ [ID]}{\{a{:}\texttt{int}\} \triangleright a : \texttt{int} \qquad \{x{:}\texttt{int}\} \triangleright 2*x : \texttt{int}}\ [*]}{\{x{:}\texttt{int}, a{:}\texttt{int}\} \triangleright a+2*x : \texttt{int}}\ [+]$$

$$\frac{\{x{:}\texttt{int}, a{:}\texttt{int}\} \triangleright a+2*x : \texttt{int}}{\{a{:}\texttt{int}\} \triangleright \texttt{fn } x \texttt{ => } a+2*x : \texttt{int -> int}}\ [FN]$$

5.2.6 Typisierungsregel für Funktionsapplikation

- **Typisierungsregel [AP] für Funktionsapplikation**

$$\frac{\Gamma \triangleright e : typ_0 \to typ_1 \qquad \Gamma_0 \triangleright e_0 : typ_0}{\vdash \Gamma \cup \Gamma_0 \triangleright e\ e_0 : typ_1}\ [AP]$$

- **Beispiel**

$$\frac{\{a:int\} \triangleright fn\ x \Rightarrow a+2*x : int \to int \qquad \{y:int\} \triangleright 5*y : int}{\{y:int,\ a:int\} \triangleright (fn\ x \Rightarrow a+2*x)(5*y) : int}\ [AP]$$

5.2.7 Typisierungsregel für lokale Deklaration

- **Typisierungsregel [LET] für lokale Deklaration**

$$\frac{\Gamma_0 \triangleright e_0 : typ_0 \quad \Gamma_1 \cup \{x: typ_0\} \triangleright e_1 : typ_1}{\vdash \Gamma_0 \cup \Gamma_1 \triangleright (\texttt{let val x=}e_0\texttt{ in }e_1\texttt{ end}) : typ_1}\ [\text{LET}]$$

- **Bemerkung**

  Möglicherweise kann x in Γ_0 gebunden auftreten.

- **Beispiel**

$$\frac{\{y:int\} \triangleright y+5 : int \quad \{x:int\} \triangleright \texttt{if x > 0 then x-1 else 0} : int}{\{y:int\} \triangleright \texttt{(let val x = y+5 in if x>0 then x-1 else 0 end)}:int}\ [\text{LET}]$$

5.2.8 Typisierungsregel für rekursive Deklaration

- **Typisierungsregel [REC] für rekursive Deklaration**

$$\frac{\Gamma_0 \cup \{\texttt{f: typ}_0\} \triangleright \texttt{e}_0 \texttt{ : typ}_0 \quad \Gamma_1 \cup \{\texttt{f: typ}_0\} \triangleright \texttt{e}_1 \texttt{ : typ}_1}{\vdash \Gamma_0 \cup \Gamma_1 \triangleright \texttt{(let val rec f=e}_0 \texttt{ in e}_1 \texttt{ end) : typ}_1} \text{ [REC]}$$

- **Bemerkung**
  - Hier muss $\texttt{typ}_0$ ein Funktionstyp und $\texttt{e}_0$ ein Funktionsausdruck sein.
  - Beachte: Allgemeinere Formen, wie `let rec f x = ...` sind nicht erfasst.
- **Beispiel**

```
{sum:int->int} ▸ fn n => if n=0 then 0
                         else n+sum(n-1) : int -> int
{sum:int->int} ▸ sum 100 : int
-------------------------------------------------------------------- [REC]
{} ▸ val rec sum = fn n => if n=0 then 0
                           else n+sum(n-1) in sum 100 : int
```

5.2.9 Polymorphie

- Die meisten Ausdrücke gestatten mehrere Typisierungen. Etwa:

```
{x:int, f:int->int}▸ f x: int
{x:int*int, f:int*int->int}▸ f x: int
{x : 'a, f: 'a -> 'a}▸ f x: 'a
{x : 'a, f: 'a -> 'b}▸ f x: 'b
```

- Die letztgenannte Typisierung ist die allgemeinstmögliche: jede andere Typisierung von `f x` lässt sich aus dieser durch Einsetzen von Typen für die Typvariablen `'a, 'b` erhalten.
- Man bezeichnet die allgemeinstmögliche Typisierung als **prinzipale Typisierung** des Ausdrucks.
- Das Einsetzen von Typen für Typvariablen nennt man **Instanzierung** der Typvariablen.

5.2.9 Polymorphie

- **Instanzierungsregel [INST]**

  $\Gamma \triangleright$ `e : typ`

  $\vdash \Gamma$`[typ`$_1$`/`α_1`, … , typ`$_n$`/`α_n`]` $\triangleright$ `e : typ[typ`$_1$`/`α_1`, … , typ`$_n$`/`α_n`]`

  wobei

  - alle α_i Typvariablen und `typ`$_i$ Typausdrücke sind und α_i **nicht** in `typ`$_i$ vorkommt,
  - `[typ`$_1$`/`α_1`, … , typ`$_n$`/`α_n`]` die **simultane Ersetzung** (oder **Substitution**) jedes Vorkommens von α_i durch `typ`$_i$ bezeichnet .

- **Beispiel**

  1. Sei der Ausdruck `{x:'a, f: 'a -> 'b}` $\triangleright$ `f x: 'b` gegeben. Dann sind
  2. `{x : 'a, f: 'a -> 'a}` $\triangleright$ `f x: 'a` und
  3. `{x : real, f: real -> int}` $\triangleright$ `f x: int`

  Hier ist (1) eine Instanz der Applikationsregel.

  (2) entsteht aus (1) durch Anwenden der Instanzierungsregel mit [`'a/'b`]

  (3) entsteht aus (1) mit [`real/'a, int/'b`].

  - Natürlich hätte man (2), (3) auch direkt mit der Applikationsregel erhalten können.

5.2.10 Typisierungsregeln für den Listentyp

- **Typisierungsaxiom [nil]**

  $\Gamma_1 \triangleright$ `nil : 'a list`

- **Typisierungsregel [cons]**

  $\Gamma_1 \triangleright e_1$ `: typ`

  $\Gamma_2 \triangleright e_2$ `: typ list`

  $\vdash \Gamma_1 \cup \Gamma_2 \triangleright e_1$ `::` e_2 `: typ list`

- **Beispiel**

$$\dfrac{\{\} \triangleright \texttt{17 : int} \qquad \dfrac{\{\} \triangleright \texttt{nil : 'a list}}{\{\} \triangleright \texttt{nil : int list}}\,[\text{INST}]}{\{\} \triangleright \texttt{17::nil : int list}}\,[\text{CONS}]$$

5.3 Typinferenz

- Das SML System kann zu beliebig vorgegebenem (geschlossenem) Ausdruck die prinzipale Typisierung automatisch berechnen (oder eine Fehlermeldung ausgeben, wenn es überhaupt keine Typisierung des Ausdrucks gibt.)
- Dies bezeichnet man als **Typinferenz**.
- **Beispiel**

  Wenn
  - `e1` den prinzipalen Typ `'a*('b->'c) -> ('b->'a)` und
  - `e2` den prinzipalen Typ `('d *'d)*(int->real)` hat,

  so berechnet man für `'a*('b->'c)` und `('d*'d)*(int->real)`
  - die allgemeinste Instanzierung σ =`['d*'d/'a,int/'b,real/'c]` mit
  - σ`('a*('b->'c)) = `σ`(('d *'d)*(int->real))`

  Dann hat `e1 e2` den prinzipalen Typ σ`('b->'a) = int -> 'd*'d`.

Typinferenz informell

- Die Typinferenz erfolgt **rekursiv über den Termaufbau**:

  Aus den prinzipalen Typen für die Teilausdrücke eines Ausdrucks bestimmt man den prinzipalen Typ für den Ausdruck, indem man auf die jeweiligen prinzipalen Typen der Teilausdrücke die **allgemeinstmögliche Instanzierung** anwendet, sodass sie zusammenpassen.

- **Beispiel**

  Wenn

  - `e1` den prinzipalen Typ `'a*('b->'c) -> ('b->'a)` und
  - `e2` den prinzipalen Typ `('d *'d)*(int->real)` hat,

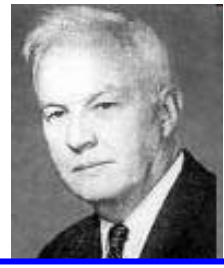
  so berechnet man für `'a*('b->'c)` und `('d*'d)*(int->real)`

  - die allgemeinste Instanzierung σ =`['d*'d/'a,int/'b,real/'c]` mit
  - σ`('a*('b->'c)) = σ(('d *'d)*(int->real))`

  Dann hat `e1 e2` den prinzipalen Typ σ`('b->'a) = int -> 'd*'d`.

Typinferenz: Geschichtliches

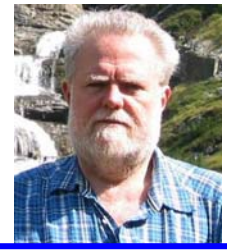
- Typinferenz
  - Haskell Curry and Robert Feys: Typinferenz für den einfach getypten Lambda-Kalkül, ca. 1958
  - Roger Hindley, The principal type-scheme of an object in Combinatory Logic, Trans AMS 146, (1969), 29-60.
  - Für ML
    - Robin Milner,"A Theory of Type Polymorphism in Programming", JCSS **17,** (1978), 348–375
    - Luis Damas; Robin Milner, "Principal type-schemes for functional programs", POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages, ACM, (1982), 207--212
- Unifikationsalgorithmus
  - John Alan Robinson, A machine-oriented logic based on the resolution principle, JACM 12,1 (1965), 23-41.
  - Alberto Martelli, Ugo Montanari: An Efficient Unification Algorithm. ACM Trans. Program. Lang. Syst. 4(2), (1982), 258-282


Haskell Curry


Robin Milner


Roger Hindley


Alberto Martelli


Ugo Montanari

5.3.1 Unifikation

- Allgemein:

  Hat
  - der geschlossene Term e_1 den prinzipalen Typ $typ_1 \texttt{->} typ_2$ und
  - der geschlossene Term e_0 den prinzipalen Typ typ_0,

  so muss man
  - die allgemeinstmögliche Instanzierung $\sigma = [t_1/\alpha_1, \ldots, t_n/\alpha_n]$ finden, sodass $\sigma(typ_1) = \sigma(typ_0)$ wird.
  - Der prinzipale Typ von `e1 e2` ist dann $\sigma(typ_2)$.
- Man darf dabei (ggf. nach Umbenennung) voraussetzen, dass typ_0 und typ_1 **keine Variablen gemeinsam** haben.
- Eine Substitution σ heißt **allgemeinster Unifikator** von typ_0 und typ_1, wenn σ die allgemeinstmögliche Instanzierung ist, so dass $\sigma(typ_0) = \sigma(typ_1)$ gilt.
- Das Auffinden des allgemeinster Unifikators heißt **Unifikation**.
- Gibt es keinen Unifikator oder hat e_1 keinen Funktionstyp, so existiert kein Typ für e_1 e_0 und ein Typfehler liegt vor.

5.3.2 Unifikationsalgorithmus

Zur Unifikation zweier Typausdrücke $\texttt{typ}_1$ und $\texttt{typ}_2$ gehe wie folgt vor:

- Wir definieren zwei Mengen M und U.
  - M stellt eine Menge von Paaren dar, die zu unifizieren sind.
  - U ist eine Menge von Substitutionen, die den allgemeinsten Unifikator von $\texttt{typ}_1$ und $\texttt{typ}_2$ aufbaut.
- **Initialisierung:** M := {($\texttt{typ}_1$, $\texttt{typ}_2$)} und U := {}
  - Anfangs enthält M nur ein Paar, das die beiden Typausdrücke enthält, deren Unifizierbarkeit überprüft werden soll.
  - Anfangs ist U leer.

Unifikationsalgorithmus

Iteration

Die Mengen M und U werden,

- so lange wie möglich und
- so lange keine erfolglose Terminierung gemeldet wird,

wie folgt verändert:

- Wähle (willkürlich) ein Paar $(\texttt{typ}_1, \texttt{typ}_2)$ aus M und verändere M und U, je nach dem, welche Gestalt $\texttt{typ}_1$ und $\texttt{typ}_2$ haben:
  1. Falls $\texttt{typ}_1$ eine Typvariable ist, dann
     a) streiche $(\texttt{typ}_1, \texttt{typ}_2)$ aus M,
     b) ersetze in M jedes Vorkommen von $\texttt{typ}_1$ durch $\texttt{typ}_2$ und
     c) füge $\texttt{typ}_2 / \texttt{typ}_1$ in U ein.
  2. Falls $\texttt{typ}_1$ eine Typkonstante ist
     a) Wenn $\texttt{typ}_2$ dieselbe Typkonstante ist, dann streiche $(\texttt{typ}_1, \texttt{typ}_2)$ aus M.
     b) Andernfalls, wenn $\texttt{typ}_2$ keine Typvariable ist, dann terminiere erfolglos.

Unifikationsalgorithmus

3. Falls typ_1 ein zusammengesetzter Typausdruck ist, der aus einem (Präfix-, Infix- oder Postfix-) Typoperator Op und den Typausdrücken $typ_{11}, \ldots, typ_{1n}$ (in dieser Reihenfolge) besteht,
   a) Wenn typ_2 ebenfalls aus demselben (Präfix-, Infix- oder Postfix-) Typoperator Op und den Typausdrücken $typ_{21}, \ldots, typ_{2n}$ (in dieser Reihenfolge) besteht, dann
      ersetze (typ_1, typ_2) in M durch die n Paare $(typ_{11}, typ_{21}), \ldots, (typ_{1n}, typ_{2n})$,
   a) Andernfalls, wenn typ_2 keine Typvariable ist, dann terminiere erfolglos.
4. Falls typ_1 keine Typvariable ist und typ_2 eine Typvariable ist, dann ersetze (typ_1, typ_2) in M durch (typ_2, typ_1).

- Ist kein Fall mehr anwendbar, dann liefere U als Unifikator von typ_1 und typ_2 und terminiere.

Anwendung des Unifikationsalgorithmus

Beispiel:

[Im Folgenden wird immer das erste Paar von M ausgewählt.]

M =	U =	
`{('a->'a list, int list->'b)}`	`{}`	Fall 3
`{('a,int list), ('a list,'b)}`	`{}`	Fall 1
`{(int list list, 'b)}`	`{(int list/'a)}`	Fall 4
`{('b, int list list)}`	`{(int list/'a)}`	Fall 1
`{}`	`{(int list/'a),` `(int list list/'b)}`	

U wird als Unifikator von 'a -> 'a list und int list -> 'b geliefert.

5.3.3 Typinferenzalgorithmus

- **Idee**
  - Um den prinzipalen Typ eine Ausdrucks zu bestimmen, berechnet man die prinzipalen Typen aller Teilterme und unifiziert die Ergebnisse gemäß der Typisierungsregeln und -Axiome.
- **Genauer**
  - Das Verfahren der Typinferenz erhält als Parameter einen Ausdruck e (mit oder ohne Typ-Constraint), dessen Typisierung überprüft oder festgestellt werden soll, und eine Menge M von Typ-Constraints.
  - Das Verfahren für einen Ausdruck e besteht aus zwei Phasen, deren Ausführungen beliebig ineinander verzahnt werden dürfen.
  - In Phase 1 werden möglichst allgemeine Typen für e und seine Teilausdrücke konstruiert.
  - Gibt es für einen Teilterm e1 mehrere Typ-Constraints werden diese in Phase 2 unifiziert.
  - Beim ersten Aufruf des Verfahrens ist M leer.

5.3.3 Typinferenzalgorithmus: Phase 1

- Hat der Ausdruck e einen (vorgegebenen) Typ-Constraint `C`, so füge `e : C` in M ein. Dann gehe wie folgt vor:
  1. Hat e die Gestalt `val x = e1` (oder `val rec x = e1`), so
     a) gebe x (bzw. `e1`) alle in M vorkommenden Typ-Constraints von `e1` (bzw. von x).
     b) Wende das Verfahren mit Ausdruck `e1` und Menge M an.
  2. Andernfalls gehe nach der Gestalt von e (gemäß den Typisierungsregeln) wie folgt vor (mit $\alpha 1$, $\alpha 2$, β Typvariablen, die weder in e noch in M vorkommen):
     a) Falls e von der Form `e1 e2` ist,
        1. so füge in M die folgenden Typ-Constraints ein:
           `e:`$\alpha 2$`, e1:`$\alpha 1$`->`$\alpha 2$`, e2:`$\alpha 1$
        2. Wende das Verfahren auf Ausdruck `e1` und Menge M an.
        3. Wende das Verfahren auxf Ausdruck `e2` und die Menge an, die sich aus der Anwendung des Verfahrens auf `e1` und M ergeben hat.

5.3.3 Typinferenzalgorithmus: Phase 1

b) Falls e von der Form (`if b then e1 else e2`) ist, so
   1. füge in M die folgenden Typ-Constraints ein:
      `e:α, b: bool, e1:α, e2:α`
   2. Wende das Verfahren auf Ausdruck b und Menge M an.
   3. Wende dann das Verfahren auf Ausdruck e1 und die Menge M1 an, die sich aus der Anwendung des Verfahrens auf b und M ergeben hat.
   4. Wende das Verfahren auf Ausdruck e2 und die Menge an, die sich aus der vorangegangenen Anwendung des Verfahrens auf e1 ergeben hat.

c) Falls e von der Form (`fn x => e1`) ist, so
   1. füge in M die folgenden Typ-Constraints ein:
      `e:α1->α2, x:α1, e1:α2`
   2. Wende das Verfahren mit Ausdruck e1 und Menge M an.

5.3.3 Typinferenzalgorithmus: Phase 2

- Kommen in M zwei Typ-Constraints `e : typ1` und `e: typ2` für denselben Ausdruck e vor, so unifiziere `typ1` und `typ2`.
  - Wenn die Unifikation von `typ1` und `typ2` erfolglos terminiert, dann melde, dass der Ausdruck e einen Typfehler enthält, und terminiere.
  - Andernfalls binde die Typvariablen, wie es sich aus der Unifikation von `typ1` und `typ2` ergibt.

- Der Algorithmus stoppt erfolgreich, wenn es in M zu jedem Teilterm von e genau einen Typ-Constraint gibt. Dann ist der allgemeinste Typ von e der Typ von e in M.

5.3.3 Typinferenzalgorithmus: Beispiele

- Sei der Ausdruck e:

```
val f = fn x => +(*(a, x), 2.0)(* d.h. a*x + 2.0 *)
```

Zerlegung des Ausdrucks	Typisierungsumgebung
`val f = fn x => +(*(a, x), 2.0)`	
	`f : 'f`
	`fn x => +(*(a,x),2.0) : 'f`
`fn x => +(*(a, x), 2.0)`	
	`f : 'f`
	`fn x => +(*(a,x),2.0) : 'f 'x->'a1`
	`x : 'x`
`+(*(a, x), 2.0)`	`f : 'f`
	`fn x => +(*(a,x),2.0) : 'f 'x->'a1`
	`x : 'x`
	`+(*(a, x), 2.0)) : 'a1`

Unifikation: `'f = 'x->'a1`

5.3.3 Typinferenzalgorithmus: Beispiele

Zerlegung des Ausdrucks	Typisierungsumgebung	
`+`	`f`	`: 'x->'a1`
	`fn x => +(*(a,x),2.0)`	`: 'x->'a1`
	`x`	`: 'x`
	`+(*(a, x), 2.0))`	`: 'a1`
	`+`	`: 'a1*'a1->'a1`
`*(a, x)`	`f`	`: 'x->'a1`
	`fn x => +(*(a,x),2.0)`	`: 'x->'a1`
	`x`	`: 'x`
	`+(*(a, x), 2.0))`	`: 'a1`
	`+`	`: 'a1*'a1->'a1`
	`*(a, x)`	`: 'a1`
`*`	`f`	`: 'x->'a1`
	`fn x => +(*(a,x),2.0)`	`: 'x->'a1`
	`x`	`: 'x`
	`+(*(a, x), 2.0))`	`: 'a1`
	`+`	`: 'a1*'a1->'a1`
	`*(a, x)`	`: 'a1 'a2`
	`*`	`: 'a2*'a2->'a2`

Unifikation: 'a2 = 'a1

5.3.3 Typinferenzalgorithmus: Beispiele

Zerlegung des Ausdrucks	Typisierungsumgebung	
a	f	: 'x->'a1
	fn x => +(*(a,x),2.0)	: 'x->'a1
	x	: 'x
	+(*(a, x), 2.0))	: 'a1
	+	: 'a1*'a1->'a1
	*(a, x)	: 'a1
	*	: 'a1*'a1->'a1
	a	: 'a1
x	f	: 'x->'a1
	fn x => +(*(a,x),2.0)	: 'x->'a1
	x	: 'x 'a1
	+(*(a, x), 2.0))	: 'a1
	+	: 'a1*'a1->'a1
	*(a, x)	: 'a1
	*	: 'a1*'a1->'a1
	a	: 'a1

Unifikation: 'x = 'a1

5.3.3 Typinferenzalgorithmus: Beispiele

Zerlegung des Ausdrucks	**Typisierungsumgebung**	
`2.0`	`f`	`: 'a1->'a1`
	`fn x => +(*(a,x),2.0)`	`: 'a1->'a1`
	`x`	`: 'a1`
	`+(*(a, x), 2.0))`	`: 'a1`
	`+`	`: 'a1*'a1->'a1`
	`*(a, x)`	`: 'a1`
	`*`	`: 'a1*'a1->'a1`
	`a`	`: 'a1`
	`2.0`	`: 'a1 real`
Unifikation: `'a1 = real`		
Ergebnis	`f`	`: real->real`
	`fn x => +(*(a,x),2.0)`	`: real->real`
	`x`	`: real`
	`+(*(a, x), 2.0))`	`: real`
	`+`	`: real*real->real`
	`*(a, x)`	`: real`
	`*`	`: real*real->real`
	`a`	`: real`

5.3.3 Typinferenzalgorithmus: Beispiele

- Sei der Ausdruck e:

```
val f = fn x => x :: nil;
```

Zerlegung des Ausdrucks	Typisierungsumgebung	
`val f = fn x => x :: nil`		
	`f`	`: 'f`
	`fn x => x :: nil`	`: 'f`
`fn x => x :: nil`		
	`f`	`: 'f`
	`fn x => x :: nil`	`: 'f 'x->'a1`
	`x`	`: 'x`
`x :: nil`	`f`	`: 'f`
	`fn x => x :: nil`	`: 'f 'x->'a1`
	`x`	`: 'x`
	`x :: nil`	`: 'a1`

Unifikation: `'f = 'x->'a1`

5.3.3 Typinferenzalgorithmus: Beispiele

Zerlegung des Ausdrucks	**Typisierungsumgebung**	
`::`	`f`	`: 'x->'a1`
	`fn x => x :: nil`	`: 'x->'a1`
	`x`	`: 'x`
	`x :: nil`	`: 'a1 'a2 list`
	`::`	`: 'a2*'a2 list->'a2 list`
Unifikation: `'a1 = 'a2 list`		
`x`	`f`	`: 'x->'a2 list`
	`fn x => x :: nil`	`: 'x->'a2 list`
	`x`	`: 'x 'a2`
	`x :: nil`	`: 'a2 list`
	`::`	`: 'a2*'a2 list->'a2 list`
Unifikation: `'x = 'a2`		
Ergebnis:	`f`	`: 'a2->'a2 list`
	`fn x => x :: nil`	`: 'a2->'a2 list`
	`x`	`: 'a2`
	`x :: nil`	`: 'a2 list`
	`::`	`: 'a2*'a2 list->'a2 list`

Zusammenfassung

Robin Milner: "*Well-typed programs do not go wrong*".

- Eine **Typaussage** hat die Form
  - $\Gamma \triangleright$ `e : typ`

  und bedeutet: Unter der Annahme, dass die Bezeichner die im Typkontext Γ angegebenen Typen haben, hat `e` den Typ `typ`."
- Typisierungsaussagen werden **formal hergeleitet** aus **Typaxiomen** und **Typisierungsregeln.**
- SML besitzt ein **polymorphes Typsystem**, bei dem jeder wohlgetypte Ausdruck einen **allgemeinsten (prinzipalen) Typ** besitzt.
- Mit Hilfe des **Hindley-Milner Typinferenzalgorithmus** kann das SML System zu beliebig vorgegebenem (geschlossenem) Ausdruck die prinzipale Typisierung automatisch berechnen (oder eine Fehlermeldung ausgeben, wenn es keine Typisierung des Ausdrucks gibt.)
- Der Typinferenzalgorithmus beruht insbesondere auf dem Verfahren der **Unifikation (von Robinson)**, bei dem für zwei Typausdrücke automatisch die allgemeinste Substitution berechnet wird, die beide Typen gleich macht (oder mit einem Fehler terminiert wird, wenn es keine solche Substitution gibt).