Kapitel 3

Typen und Prozeduren

Die in Kapitel 1 vorgestellte Teilsprache von Standard ML realisiert im Zusammenhang mit Typen und Prozeduren eine Vielzahl von interessanten Konzepten, die wir in diesem Kapitel näher betrachten werden.

Zu einer Programmiersprache gehören syntaktische und semantische Objekte. Ausdrücke und Deklarationen sind Beispiele für syntaktische Objekte. Die bei der Auswertung eines Ausdrucks berechneten Werte sind Beispiele für semantische Objekte. Die syntaktischen und semantischen Objekte einer Programmiersprache sind als mathematische Objekte zu verstehen und können mit den in Kapitel 2 eingeführten Konzepten beschrieben werden.

Beim Programmieren werden die syntaktischen Objekte einer Programmiersprache durch Zeichenfolgen beschrieben. Wenn der Interpreter den Wert eines Ausdrucks ausgibt, beschreibt er diesen ebenfalls durch eine Zeichenfolge. Die Unterscheidung zwischen Objekten (als mathematische Objekte) und ihrer Repräsentation (durch Zeichenreihen) ist eine wesentliche Voraussetzung für das Verständnis von Programmiersprachen.

Ein Interpreter verarbeitet eine Eingabe in vier Phasen:

1. *Lexikalische Analyse.* Es wird überprüft, ob die eingegebene Zeichenfolge in eine Folge von Wörtern der Sprache zerlegt werden kann.

2. *Syntaktische Analyse.* Es wird überprüft, ob die von der lexikalischen Analyse gelieferte Wortfolge eine Phrase der Sprache beschreibt. Wenn dies der Fall ist, wird die beschriebene Phrase als syntaktisches Objekt konstruiert. Statt von syntaktischer Analyse spricht man auch von *Parsing*.

3. *Semantische Analyse.* Es wird überprüft, ob die von der syntaktischen Analyse gelieferte Phrase die Regeln der statischen Semantik einhält. Statt von semantischer Analyse spricht man auch von *Elaboration*.

4. *Ausführung.* Wenn die Analysephasen keinen Fehler gefunden haben, wird die von der syntaktischen Analyse gelieferte Phrase ausgeführt. Statt von Ausführung spricht man auch von *Auswertung* oder von *Evaluation*.

Man unterscheidet zwischen statischen und dynamischen Aspekte von Programmiersprachen. Die die Analysephasen betreffenden Aspekte bezeichnet man als *statisch*. Die die Ausführung betreffenden Aspekte bezeichnet man als *dynamisch*.

3.1 Typen und Typdisziplin

Jedem Typ sind Werte zugeordnet. Beim Typ `int` sind dies ganze Zahlen, bei `real` rationale Zahlen, bei `unit` das leere Tupel, und bei `bool` die ganzen Zahlen 0 und 1. Die Konstanten `false` und `true` dienen als Namen für die Werte 0 und 1.

Da die Booleschen Werte ganze Zahlen sind, würde man vermuten, dass Ausdrücke wie

```
1-true
```

zulässig sind. Solche Ausdrücke sind jedoch unzulässig, da sie die Typregeln von Standard ML verletzen:

1. Die Konstante `1` hat den Typ `int`.
2. Die Konstante `true` hat den Typ `bool`.
3. Die Operation ⟨`-`⟩ hat die Typen

   ```
   int * int -> int
   real * real -> real
   ```

Der Ausdruck `1-true` wäre nur dann zulässig, wenn die Operation ⟨`-`⟩ zusätzlich den Typ

```
int * bool -> int
```

hätte, was jedoch nicht der Fall ist. Dieses Argument erklärt auch, warum Ausdrücke wie `1.0-2` unzulässig sind.

Typen und die damit verbundene *Typdiziplin* sind ein wichtiger Aspekt von Programmiersprachen. Typen legen eine *Interpretation* für ihre Werte fest. Beispielsweise legt der Typ `bool` fest, dass seine Werte als Boolesche Werte zu interpretieren sind (und nicht als ganze Zahlen). Die Typdiziplin einer Programmiersprache sorgt dafür, dass auf Werte nur solche Operationen angewendet werden können, die die Werte richtig interpretieren. Bevor ein Interpreter eine eingegebene Phrase

ausführt, stellt er zunächst sicher, dass diese Phrase die Regeln der Typdiziplin einhält. Der Interpreter erzwingt also die Einhaltung der Typdisziplin.

Viele der üblichen Programmierfehler führen dazu, dass die Regeln der Typdisziplin verletzt werden. Da die Einhaltung der Typdisziplin automatisch überprüft werden kann, können diese Programmierfehler automatisch gefunden werden. Das automatische Finden von Programmierfehlern mithilfe einer Typdisziplin ist bei der Entwicklung größerer Programme außerordentlich hilfreich.

Wir beschreiben jetzt eine vereinfachte Variante der Typdisziplin von Standard ML, die als *monomorphe Typdisziplin* bezeichnet wird:

1. Jeder Konstante und jedem gebundenen Bezeichner ist genau ein Typ zugeordnet.
2. Jedem Operationssymbol (zum Beispiel ⟨`+`⟩ und ⟨`-`⟩) sind endlich viele Typen zugeordnet. Operationssymbole, denen mehr als ein Typ zugeordnet ist (zum Beispiel ⟨`+`⟩ und ⟨`-`⟩), werden als *überladen* bezeichnet.
3. Für Operationsausdrücke wird die folgende Typregel verwendet:

$$\frac{a_1 : t_1 \qquad o : t_1 * t_2 \to t \qquad a_2 : t_2}{a_1\ o\ a_2\ :\ t}$$

4. Für Konditionale wird die folgende Typregel verwendet:

$$\frac{a_1 : \texttt{bool} \qquad a_2 : t \qquad a_3 : t}{\texttt{if}\ a_1\ \texttt{then}\ a_2\ \texttt{else}\ a_3\ :\ t}$$

   Die Konsequenz und die Alternative eines Konditional müssen also denselben Typ haben.

5. Für Tupelausdrücke wird die folgende Typregel verwendet:

$$\frac{a_1 : t_1 \qquad \cdots \qquad a_n : t_n}{(a_1, \ldots, a_n)\ :\ t_1 * \cdots * t_n}$$

6. Für Prozeduranwendungen wird die folgende Typregel verwendet:

$$\frac{p : t_1 \to t_2 \qquad a : t_1}{p\ a\ :\ t_2}$$

7. Die obigen Regeln sorgen dafür, dass jedem zulässigen Ausdrück genau ein Typ zugeordnet werden kann.
8. Wenn die Auswertung eines Ausdrucks, dem der Typ t zugeordnet wurde, einen Wert liefert, handelt es sich immer um einen Wert des Typs t. Diese wichtige Eigenschaft der Typdiziplin wird als *Typsicherheit* bezeichnet.

Betrachten Sie die folgende Interaktion mit dem Interpreter:

```
1<2
true : bool
```

Der vom Interpreter ausgegebene Typ `bool` ist der Typ, der dem Ausdruck `1<2` zugeordnet wurde. Dieser Ausdruck wertet zu der Zahl 1 aus. Da der Interpreter weiß, dass dieser Wert gemäß dem Typ `bool` zu interpretieren ist, stellt er ihn mit der Konstante `true` dar.

Typen können nicht mit der Menge ihrer Werte identifiziert werden. Typen legen zusätzlich eine Interpretation für ihre Werte fest. Die Situation ist vergleichbar mit Prozeduren, die nicht mit den durch sie berechneten Funktionen identifiziert werden können. Prozeduren legen zusätzlich einem Algorithmus fest.

3.2 Prozeduren sind Werte

Prozeduren sind Werte. Das bedeutet, dass Prozeduren als Komponenten von Tupeln und als Argumente und Ergebnisse von Prozeduren vorkommen können. Die Situation ist analog zu Funktionen, die so wie andere mathematische Objekte auch als Komponenten von Tupeln, als Elemente von Mengen und als Elemente des Definitions- und Wertebereichs von Funktionen auftreten können. Hier sind Beispiele:

```
fun f(x:int) = 2*x
val f : int -> int

(2,f)
(2, fn) : int * (int -> int)

fun g(f:int->int, x:int) = f (f x)
val g : (int -> int) * int -> int

g(f,5)
20 : int

fun h(x:int) = (x,f)
val h : int -> int * (int -> int)

h 7
(7, fn) : int * (int -> int)

#2(h 7) 12
24 : int
```

Der Interpreter stellt prozedurale Werte bei der Ausgabe immer mit dem Wort `fn` dar.

Prozeduren mit prozeduralen Argumenten oder Ergebnissen bezeichnet man als *höherstufige Prozeduren*. Später werden wir sehen, dass höherstufige Prozeduren eine Reihe von interessanten Programmiertechniken ermöglichen.

Analog zur Lambda-Notation für Funktionen kann man Prozeduren mit speziellen Ausdrücken beschreiben, die als *Abstraktionen* bezeichnet werden:

```
fn x:int => x*x
fn : int -> int

(fn x:int => x*x) 7
49 : int

fn (x:int, y:int) => x*y
fn : int * int -> int

(fn (x:int, y:int) => x*y) (4,7)
28 : int
```

Abstraktionen sind Ausdrücke, die zu Prozeduren auswerten. Eine Abstraktion beschreibt eine Prozedur ohne ihr einen Namen zu geben. Die Bindung an einen Namen kann durch eine Wertdeklaration erfolgen:

```
val kreisflaeche = fn r:real => r*r*Math.pi
val kreisflaeche : real -> real

kreisflaeche 2.0
12.5663706144 : real
```

Rekursive Prozeduren kann man wegen der Selbstreferenz nicht unmittelbar mit Abstraktionen beschreiben. Die gelingt jedoch mit *rekursiven Wertdeklarationen*:

```
val rec f = fn n:int => if n<1 then 1 else f(n-1)
val f : int -> int
```

Das Schlüsselwort `rec` sagt, dass der Bezeichner `f` mithilfe einer rekursiven Deklaration an einen Wert gebunden werden soll. Die uns bereits bekannten Prozedurdeklarationen mit `fun` sind als Abkürzungen für rekursive Wertdeklarationen mit Abstraktionen zu verstehen.

Ein Beispiel für prozedurale Argumente

Wir zeigen jetzt mit einem Beispiel, dass die Verwendung von Prozeduren als Argumente von Prozeduren für die Programmierung sinnvoll ist.

Sei eine Funktion $f \in \mathbb{N} \to \mathbb{Z}$ gegeben. Für $n \geq 0$ definieren wir die Σ-*Notation* wie folgt:

$$\sum_{i=1}^{n} f(i) \stackrel{\text{def}}{=} \text{if } n = 0 \text{ then } 0 \text{ else } f(1) + f(2) + \cdots + f(n)$$

Mit der Σ-Notation können wir eine Funktion

$$sum \in (\mathbb{N} \to \mathbb{Z}) \times \mathbb{N} \;\to\; \mathbb{Z}$$
$$sum(f, n) = \sum_{i=1}^{n} f(i)$$

definieren. Diese können wir mit der rekursiven Prozedur

```
fun sum(f:int->int, n:int) =
    if n<1 then 0 else sum(f,n-1) + f n

val sum : (int -> int) * int -> int
```

berechnen. Die Summe der ersten 10 Quadratzahlen erhalten wir wie folgt:

```
sum(fn i:int => i*i, 10)
385 : int
```

Die Summe der ersten 15 natürlichen Zahlen bekommen wir mit

```
sum(fn i:int => i, 15)
120 : int
```

Schließlich bekommen wir die Summe $\sum_{i=1}^{23} 2i^2 - 3i$ mit

```
sum(fn i:int => 2*i*i-3*i, 23)
7820 : int
```

3.3 Kartesische und kaskadierte Prozeduren

Wir wollen jetzt ganzzahlige Addition als eine Prozedur

```
plus : int * int -> int
```

zur Verfügung stellen. Das ist nicht schwer:

```
fun plus (x:int, y:int) = x+y
val plus : int * int -> int

plus(3,7)
10 : int
```

Prozeduren, deren Argumenttyp ein Produkttyp $t_1 * \cdots * t_n$ ist, bezeichnet man als *kartesisch*. Prozeduren deren Ergebnistyp ein Pfeiltyp $t_1 \to t_2$ ist, bezeichnet man als *kaskadiert*. Wir haben gerade gesehen, wie man die ganzzahlige Addition mithilfe einer kartesischen Prozedur zur Verfügung stellen kann. Alternativ haben wir die Möglichkeit, ganzzahlige Addition mithilfe einer kaskadierten Prozedur

```
val plus : int -> (int -> int)
```

zur Verfügung zu stellen. Dann ist `plus` eine Prozedur, die nur eine Zahl als Argument bekommt. Wenn wir diese Version von `plus` auf eine Zahl x anwenden, erhalten wir als Ergebnis eine Prozedur, in die x „eingebaut“ ist. Die Anwendung dieser Hilfsprozedur auf eine Zahl y liefert dann die Summe $x + y$. Die kaskadierte Version von `plus` kann wie folgt deklariert und angewendet werden:

```
fun plus (x:int) (y:int) = x+y
val plus : int -> (int -> int)

plus 3 7
10 : int

val p = plus 7
val p : int -> int

p 8
15 : int
```

Der Ausdruck

```
plus 3 7
```

wird als

```
(plus 3) 7
```

interpretiert. Die Prozedur `plus` wird also zuerst auf `3` angewendet. Die so berechnete Hilfsprozedur wird dann auf `7` angewendet. Allgemeiner gilt, dass eine Folge

$$a_1 \;\; a_2 \;\; a_3$$

von drei Ausdrücken als eine geschachtelte Anwendung

$$(a_1 \;\; a_2) \;\; a_3$$

interpretiert wird. Die innere Anwendung $a_1 \;\; a_2$ muss dabei eine Prozedur liefern. Hier ist ein Beispiel:

```
(fn x:int => fn y:int => x*y) 3 (4+7)
33 : int
```

Bei kaskadierten Pfeiltypen kann man ebenfalls Klammern weglassen. Beispielsweise wird

$$t_1 \to t_2 \to t_3$$

als eine Abkürzung für

$$t_1 \to (t_2 \to t_3)$$

interpretiert. Wir stellen also fest, dass Prozeduranwendung nach links und der Typkonstruktor $\rightarrow$ nach rechts klammert.

Die kartesische und die kaskadierte Variante von `plus` können auch mithilfe von Abstraktionen formuliert werden:

```
val plus = fn (x:int, y:int) => x+y
val plus : int * int -> int

val plus = fn x:int => fn y:int => x+y
val plus : int -> int -> int
```

Die kaskadierte Formulierung von mehrstelligen Funktionen oder Prozeduren wird im Englischen als „curried version“ bezeichnet. Das Kunstwort „curried“ bezieht sich dabei auf den Logiker Haskell B. Curry, der die kaskadierte Darstellung popularisierte. Die Entdeckung der kaskadierten Formulierung wird dem Logiker M. Schönfinkel zugesprochen.

Wir haben bereits die kartesische Version einer Prozedur `sum` zur Berechnung der Summe $\sum_{i=1}^{n} f(i)$ gesehen. Hier ist eine kaskadierte Version:

```
fun sum (f: int->int) (n:int) =
    if n<1 then 0 else sum f (n-1) + f n

val sum : (int -> int) -> int -> int
```

Damit können wir den Wert von

$$\sum_{i=0}^{5} i^2 + \sum_{i=0}^{17} i^2$$

beispielsweise wie folgt berechnen:

```
val qs = sum (fn i:int => i*i)
val qs : int -> int

qs 5 + qs 17
1840 : int
```

3.4 Polymorphe Prozeduren

Sei X eine Menge, $f \in X \rightarrow X$ und $n \geq 0$. Wir definieren die Funktion f^n rekursiv wie folgt:

$$f^n \in X \rightarrow X$$
$$f^n(x) = \text{if } n = 0 \text{ then } x \text{ else } f^{n-1}(f(x))$$

Das Ergebnis von $f^n(x)$ erhält man also dadurch, dass man die Funktion f n-mal hintereinander auf x anwendet.

Dieser Idee folgend wollen wir jetzt eine kaskadierte Prozedur

```
iter : (t -> t) -> int -> t -> t
```

deklarieren, die zu einer Prozedur f, einem $n \geq 0$ und einem Wert x den Wert liefert, den man durch n-maliges Anwenden von f aus x erhält. Dabei soll `iter` für jeden Typ t anwendbar sein. Dies gelingt mit der folgenden Deklaration:

```
fun iter (f:'a->'a) (n:int) (x:'a) =
    if n<1 then x else iter f (n-1) (f x)

val iter : ('a -> 'a) -> int -> 'a -> 'a
```

Dabei ist `'a` eine sogenannte *Typvariable*. Hier sind Bespiele für die Anwendung von `iter`:

```
iter (fn x:int => x*x) 4
fn : int -> int

iter (fn x:int => x*x) 3 2
256 : int

iter (fn x:int => x*x) 4 2
65536 : int

iter (fn x:int => x*x) 5 2
! Uncaught exception: Overflow

iter (fn x:real => x*x) 5 2.0
4294967296.0 : real

iter (fn x:bool => not x) 3 true
false : bool
```

Prozeduren, die auf mehr als einen Typ anwendbar sind, bezeichnet man als *polymorph*. Dagegen bezeichnet man Prozeduren, die nur auf einen Typ anwendbar sind, als *monomorph*.

Wir erweitern jetzt die in Abschnitt 3.1 formulierte monomorphe Typdisziplin um die Möglichkeit, Prozeduren polymorph zu typisieren. Die so erweiterte Typdisziplin bezeichnen wir als *polymorphe Typdisziplin*. Als Leitbeispiel deklarieren wir eine polymorphe Prozedur, die die sogenannte Identitätsfunktion berechnet:

```
fun id(x:'a) = x
val id : 'a -> 'a
```

Diese Deklaration ordnet dem Bezeichner `id` das folgende *Typschema* zu:

```
∀ 'a ('a -> 'a)
```

Ein Typschema repräsentiert alle Typen, die sich als *Instanzen* des Schemas erhalten lassen. Diese erhält man, indem man die Typvariablen des Schemas durch Typen ersetzt. Das obige Typschema hat unter anderem die folgenden Instanzen:

```
int -> int
real -> real
'b -> 'b
(int * int) -> (int * int)
('c * 'c) -> ('c * 'c)
```

Im Rahmen der polymorphen Typdisziplin können auch Typvariablen als Typen verwendet werden. Typvariablen sind spezielle Bezeichner, die mit einem einleitenden Hochkomma geschrieben werden, zum Beispiel `'a`, `'b`, `'c` und `'typvariable`. Die Typvariablen `'a`, `'b` und `'c` liest man als *alpha*, *beta* und *gamma*.

Jedes Typschema hat unendlich viele Instanzen. Das liegt daran, dass jedes Typschema mindestens eine instanziierbare Typvariable enthält und mithife des Pfeils unendliche viele Typen konstruiert werden können.

Ein Bezeichner heißt *monomorh* genau dann, wenn ihm genau ein Typ zugeordnet ist, und *polymorph*, wenn ihm ein Typschema zugeordnet ist. Polymorphe Bezeichner können nur mit Deklarationen eingeführt werden. Die Argumentvariablen von Abstraktionen und Prozedurdeklarationen sind immer monomorphe Bezeichner.

Ein polymorpher Bezeichner kann mit jedem der ihm durch sein Typschema zugeordneten Typen benutzt werden. Folglich kann Ausdrücken, die einen polymorphen Bezeichner enthalten, im Allgemeinen mehr als ein Typ zugeordnet werden. Gegeben die Deklaration

```
fun id(x:'a) = x
val id : 'a -> 'a
```

kann beispielsweise dem Ausdruck

```
id id
```

(die Anwendung von `id` auf `id`) jeder Typ $t \to t$ zugeordnet werden.

Ein Ausdruck heißt *monomorph*, wenn ihm nur ein Typ zugeordnet werden kann. Dagegen heißt ein Ausdruck, dem mehrere Typen zugeordnet werden können, *polymorph*.

Die Typdisziplin von Standard ML ist so entworfen, dass alle Typen eines polymorphen Ausdrucks immer durch nur ein Typschema beschrieben werden können.

Ein Ausdruck heißt *expansiv*, wenn er eine Prozedur- oder eine Operations-

anwendung enthält, die nicht innerhalb einer Abstraktion steht. Beispielsweise sind `id 3` und `4+5` expansive Ausdrücke. Dagegen ist die Abstraktion `fn x:int => (id x, x+5)` ein nicht-expansiver Ausdruck.

Ein durch eine Deklaration

```
val x = a
```

gebundener Bezeichner *x* wird wie folgt typisiert:

1. *Monomorphe Deklaration:* Wenn *a* ein monomorpher Ausdruck mit dem Typ *t* ist, wird *x* monomorph mit dem Typ *t* typisiert.
2. *Polymorphe Deklaration:* Wenn *a* ein polymorpher Ausdruck ist, der nicht expansiv ist, wird *x* polymorph mit dem Typschema typisiert, dass alle Typen von *a* beschreibt.
3. *Ambige Deklaration:* Wenn *a* ein polymorpher und expansiver Ausdruck ist, wird *x* monomorph mit einem der dem Ausdruck *a* zugeordeneten Typen typisiert. Dabei wird der Typ so gewählt, dass er mit den nachfolgenden Verwendungen des Bezeichners *x* verträglich ist (wenn dies möglich ist).[1]

Die Spezialbehandlung von ambigen Deklarationen ist aufgrund von Sprachkonstrukten erforderlich, die wir erst später einführen werden (es handelt sich um die sogenannte Referenzen, siehe Abschnitt 13.8). Die folgenden Bespiele zeigen, wie der Interpreter mit ambigen Deklarationen umgeht:

```
fun id (x:'a) = x
val id : 'a -> 'a

val f = id id
val f : 'b -> 'b
! Warning: Value polymorphism, free type variable 'b

f 5
5 : int
! Warning: free type variable 'b has been instantiated to int

f 5.0
! Type clash: expression of type real cannot have type int
```

Die erste Deklaration ist polymorph. Die zweite Deklaration ist ambig. Da der Interpreter den Typ von `f` noch nicht festlegen kann, beschreibt er ihn durch die Typvariable `'b`. Die Anwendung von `f` auf `5` sorgt dafür, dass `f` den Typ `int->int` bekommt. Damit ist die nachfolgende Anwendung `f 5.0` unzulässig. Da die Prozedur `id` im Gegensatz zu `f` polymorph typisiert ist, kann sie auf beliebige Typen angewendet werden:

[1] Ambig bedeutet mehrdeutig. Die Standard ML Definition spricht bei ambigen Deklarationen von „value polymorphism“.

```
id 5
5 : int

id 5.0
5.0 : real
```

Bei rekursiven Deklarationen

```
val rec x = a
```

gibt es nur den monomorphen und den polymorphen Fall, da die rechte Seite von rekursiven Wertdeklarationen grundsätzlich eine Abstraktion sein muss. Da Prozedurdeklaration Abkürzungen für rekursive Wertdeklarationen sind, kann auch hier der ambige Fall nicht auftreten.

Die Abstraktion

```
fn x : 'a => x
```

ist ein Beispiel für einen polymorphen Ausdruck, der keinen polymorphen Bezeichner enthält (Argumentvariablen sind immer monomorph typisiert). Für den Ausdruck existieren mehrere Typen, da der Typ der Argumentvariable mit einer Typvariable `'a` angegeben ist, die beliebig instanziiert werden darf. Das Typschema

```
∀ 'a ('a -> 'a)
```

beschreibt alle Typen des Ausdrucks.

Schließlich betrachten wir noch ein extremes Beispiel:

```
fun f (x:int) = f x
val f : int -> 'a
```

Die Prozedur `f` divergiert für alle Argumente. Der Bezeichner `f` wird polymorph mit dem Schema

```
∀ 'a (int -> 'a)
```

typisiert, da als Ergebnistyp von `f` jeder Typ zulässig ist.

3.5 Typinferenz

In Standard ML ist es möglich, die Typconstraints[2] für die Argumentvariablen in Prozedurdeklarationen und Abstraktionen wegzulassen, da diese automatisch hergeleitet werden können. Das automatische Herleiten der fehlenden Typconstraints wird als *Typinferenz* bezeichnet. Hier sind Beispiele:

[2] Constraints (englisch) sind Bedingungen.

```
fun kon(x,y) = if x then y else false
val kon : bool * bool -> bool

fun sum f n = if n<1 then 0 else sum f (n-1) + f n
val sum : (int -> int) -> int -> int

val f = sum (fn x => x*x)
val f : int -> int
```

Im Zusammenhang mit überladenen Operationssymbolen liefert Typinferenz nicht immer das gewünschte Ergebnis:

```
fun plus(x:real, y:real) = x+y
val plus : real * real -> real

fun plus(x,y) = x+y
val plus : int * int -> int
```

Typinferenz gibt also `int` den Vorzug, wenn der Kontext eines überladenen Operationssymbols mehrere Möglichkeiten zulässt.

Typinferenz berechnet immer die *allgemeinsten Typconstraints* (außer bei arithmetischen Typen, da sie dort im Zusammenhang mit überladenen Operationssymbolen nicht immer existieren, siehe oben):

```
fun f(x:int, y:int) = (2*x, y)
val f : int * int -> int * int

fun f(x,y) = (2*x, y)
val f : int * 'a -> int * 'a
```

Im Folgenden werden wir Typconstraints der Kürze halber oft weglassen. In der Praxis kann es aber durchaus sinnvoll sein, Typconstraints anzugeben, da deren Präsenz die Lesbarkeit eines Programmes verbessern kann.

Der Programmier hat auch die Möglichkeit, Typconstraints für die Ergebnisse von Prozeduren anzugeben:

```
fun f (x:int) : int->int = (fn y => x+y)
val f : int -> int -> int
```

Wenn man größere Programme schreibt, macht man fast immer Fehler. Viele dieser Fehler treten als Typfehler in Erscheinung, die vom Interpreter automatisch aufgedeckt werden. Es ist nicht immer einfach, aufgrund der Fehlermeldungen des Interpreters die genaue Fehlerstelle zu finden. Explizite Typconstraints können die Qualität der Fehlermeldungen erheblich verbessern, da sich dann Fehler in einer Prozedurdeklaration direkt bei der Deklaration diagnostizieren lassen. Wenn dagegen Typconstraints weggelassen werden, kann es sein, dass der Interpreter eine Prozedur mit einem nicht intendierten Typ typisiert und den Fehler erst später bei einer Anwendung der Prozedur findet.

3.6 Op-Ausdrücke

Op-Ausdrücke sind Abkürzungen für Abstraktionen, die Operationen als kartesische Prozeduren zur Verfügung stellen. Hier sind Beispiele:

```
op+       ⟹    fn (x,y) => x+y
op<       ⟹    fn (x,y) => x<y
op=       ⟹    fn (x,y) => x=y
op div    ⟹    fn (x,y) => x div y
op ~      ⟹    fn x => ~x
```

Da die meisten Operationssymbole überladen sind, ist das Weglassen der Typconstraints für die Argumentvariablen der Abstraktionen wesentlich:

```
op+(8,9)
17 : int

op+(8.0, 9.0)
17.0 : real

op-(op div (12,3), 7)
~3 : int

op+
fn : int * int -> int

op+ : real * real -> real
fn : real * real -> real
```

Das letzte Beispiel zeigt, wie man mit einem Typconstraint erzwingen kann, dass ein Op-Ausdruck den richtigen Typ hat.

3.7 Typen und Gleichheit

In Standard ML ist der Gleichheitstest nicht für alle Typen verfügbar. Typen, für deren Werte ein Test auf Gleichheit mit `=` möglich ist, heißen *Typen mit Gleichheit.* Die Typen `int`, `bool` und `unit` sind Beispiele für Typen mit Gleichheit. Dagegen sind die Pfeiltypen $t_1 \to t_2$ Typen ohne Gleichheit.[3] Auf Produktypen $t_1 * \cdots * t_n$ ist der Gleichheitstest genau dann zulässig, wenn er auf allen Komponententypen $t_1, \ldots, t_n$ zulässig ist.

Um die Unterscheidung zwischen Typen mit und ohne Gleichheit in Typschemata ausdrücken zu können, gibt es zwei Arten von Typvariablen. Während Typvariab-

[3] Es gibt keine wirklich zwingenden Gründe, das Testen von Gleichheit für Prozeduren zu verbieten. Solche Gleichheitstests sind aber normalerweise auch nicht sinnvoll. Später werden wir sogenannte abstrakte Typen kennenlernen, bei denen das Verbieten des Gleichheitstests ganz offensichtlich sinnvoll ist.

len mit nur einem Hochkomma (zum Beispiel `'a`) zu beliebigen Typen instanziiert werden können, können Typvariablen mit zwei Hochkommas (zum Beispiel `''a`) nur zu Typen mit Gleichheit instanziiert werden. Hier sind Beispiele:

```
2=3
false : bool

(fn x => x) = (fn x => x)
! Type clash: 'b -> 'c is not an equality type

op=
fn : ''a * ''a -> bool
```

3.8 Bezeichnerbindung

Programmiersprachen verwenden Bezeichner als Namen für Objekte, die während der Ausführung einer Phrase berechnet werden. Hier ist ein Beispiel:

```
val f = fn x => x*x
val f : int -> int
```

Diese Deklaration verwendet die Bezeichner `f` und `x`. Der Bezeichner `x` wird als Argumentvariable einer Abstraktion benutzt. Die Ausführung der Deklaration bindet den Bezeichner `f` an eine Prozedur. Die Ausführung der Anwendung

```
f 5
25 : int
```

bindet die Argumentvariable `x` an die Zahl 5.

Die gerade besprochenen Bezeichnerbindungen bezeichnet man als *dynamische Bindungen*. Zusätzlich gibt es *statische Bezeichnerbindungen*, die ein Interpreter während der semantischen Analyse einer Phrase berechnet. Die Analyse der Deklaration

```
val f = fn x => x*x
val f : int -> int
```

bindet den Bezeichner `f` an den Typ `int->int` und den Bezeichner `x` an `int`.

Neben dynamischen und statischen Bindungen gibt es sogenannte *lexikalische Bindungen*. Lexikalische Bindungen beschreiben den strukturellen Rahmen für statische und dynamische Bindungen. Dazu betrachten wir den Ausdruck

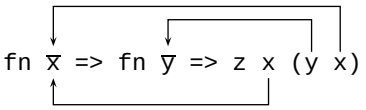
```
fn x => fn y => z x (y x)
```

In diesem Ausdruck treten drei Bezeichner auf: `x`, `y` und `z`. Der Bezeichner `x` kommt insgesamt dreimal, `y` zweimal und `z` einmal vor. Wir sprechen von den *Auftreten eines Bezeichners*. Für die Auftreten eines Bezeichners in einer Phrase

gibt es zwei Möglichkeiten: *definierend* und *benutzend*. Definierende Auftreten führen eine Bindung ein und benutzende Auftreten benutzen eine bestehende Bindung. In unserem Beispiel gibt es zwei definierende und vier benutzende Bezeichnerauftreten. Wenn wir die definierenden Bezeichnerauftreten durch Überstreichen markieren, bekommen wir die folgende Darstellung:

```
fn x̄ => fn ȳ => z x (y x)
```

Eine lexikalische Bindung bindet in einer Phrase ein benutzendes Auftreten eines Bezeichners *x* an ein definierendes Auftreten von *x*. Für unser Beispiel können wir die lexikalischen Bindungen grafisch wie folgt darstellen:



Jeder Pfeil stellt eine lexikalische Bindung dar. Das benutzende Auftreten von *z* kann innerhalb des Ausdrucks keinem definierenden Auftreten von *z* zugeordnet werden.

Benutzende Bezeichnerauftreten, die innerhalb der betrachteten Phrase lexikalisch ungebunden sind (so wie das Auftreten von *z* im Beispiel), werden als *freie Auftreten* bezeichnet. Wir sagen, dass ein Bezeichner *x* in einer Phrase *frei vorkommt*, wenn es mindestens ein freies Auftreten von *x* in der Phrase gibt.

Bei der semantischen Analyse einer eingegebenen Phrase prüft der Interpreter, dass in ihr nur solche Bezeichner frei vorkommen, für die bereits statische und dynamische Bindungen existieren.

Da die grafische Darstellung von lexikalischen Bindungen für größere Ausdrücke schnell unhandlich wird, weichen wir auf eine textuelle Darstellung aus:

```
fn x̄1 => fn ȳ1 => z x1 (y1 x1)
```

Alle Bezeichnerauftreten einer Bindungsgruppe werden dabei mit demselben Index markiert. Dagegen bekommen die freien Bezeichner keinen Index. Wenn derselbe Bezeichner mehrere definierende Auftreten hat, werden diese Auftreten mit verschiedenen Indizes versehen. Beispielsweise können die lexikalischen Bindungen des Ausdrucks

```
fn x => fn y => (fn y => (fn x => z x y) x) y
```

wie folgt dargestellt werden:

```
fn x̄1 => fn ȳ1 => (fn ȳ2 => (fn x̄2 => z x2 y2) x1) y1
```

Abbildung 3.1 zeigt die lexikalischen Bindungen eines Let-Ausdrucks mit mehreren Deklarationen. In diesem Ausdruck treten die Bezeichner `x` und `y` frei auf.

```
let
    val x = 2*x
    val x = 2*x
    fun f x = if x<2 then x else f(x-1)
    val f = fn x => f x
in
    f x - y
end

let
    val x1 = 2*x
    val x2 = 2*x1
    fun f1 x3 = if x3<2 then x3 else f1(x3-1)
    val f2 = fn x4 => f1 x4
in
    f2 x2 - y
end
```

Abbildung 3.1: Die lexikalischen Bindungen eines Let-Ausdrucks

Im Zusammenhang mit einer Abstraktion

```
fn x => a
```

sagt man, dass der *Gültigkeitsbereich* oder *Skopus* des definierenden Auftretens von *x* der Ausdruck *a* ist. Damit meint man, dass sich alle benutzenden Auftreten von *x*, die lexikalisch an das definierende Auftreten von *x* gebunden sind, innerhalb des Ausdrucks *a* befinden müssen.

Ein Phrase *p* heißt *bereinigt*, wenn in *p* kein Bezeichner mehr als ein definierendes Auftreten hat, und wenn kein Bezeichner, der in *p* ein definierendes Auftreten hat, in *p* frei vorkommt. Zu jeder Phrase kann man durch *konsistente Umbenennung* von Bezeichnerauftreten bedeutungsgleiche bereinigte Phrasen angeben. Beispielsweise kann man aus dem Ausdruck

```
fn x => fn y => (fn y => (fn x => z x y) x) y
```

durch konsistente Umbenennung den folgenden bedeutungsgleichen bereinigten Ausdruck erhalten:

```
fn x => fn y => (fn u => (fn v => z v u) x) y
```

Hier sind einige Experimente zur Bezeichnerbindung, die Sie jetzt verstehen sollten:

```
val x = 2*x
! Unbound value identifier: x
```

```
val x = 3
val x = 3 : int
```

```
val x = 2*x
val x = 6 : int

fun f y = x+y
val f : int -> int

f 5
11 : int

val x = 2*x
val x = 12 : int

f 5
11 : int
```

Die letzte Interaktion zeigt, dass die an `f` gebundene Prozedur sich nicht ändert, wenn der in ihrer Deklaration vorkommende Bezeichner `x` später redeklariert wird. Grundsätzlich gilt, dass eine Prozedur, so wie andere Werte auch, unveränderlich ist.

3.9 Prozeduren und Auswertungsprotokolle

In Kapitel 1 haben wir die Auswertung von Ausdrücken mithilfe von Auswertungsprotokollen beschrieben. Dabei sind wir stillschweigend davon ausgegangen, dass wir Werte durch Ausdrücke darstellen können. Da wir jetzt auch Prozeduren als Werte ansehen, stellt sich die Frage, wie Prozeduren durch Ausdrücke dargestellt werden können.

Ein Ausdruck heißt *geschlossen*, wenn er keine freien Bezeichnerauftreten enthält, und *offen*, wenn er mindestens ein freies Bezeichnerauftreten enthält. Offene Ausdrücke sind unvollständige Beschreibungen, die nur im Kontext von externen Bezeichnerbindungen interpretiert werden können. Dagegen sind geschlossene Ausdrücke vollständige Beschreibungen, die ohne externe Bezeichnerbindungen interpretiert werden können.

In Auswertungsprotokollen werden Werte immer mit geschlossenen Ausdrücken dargestellt.

Nichtrekursive Prozeduren werden durch geschlossene Abstraktionen dargestellt. Beispielsweise führen die Deklarationen

```
val x = 5
fun f y = x+y
fun g x = 2 * f x
```

die folgenden dynamischen Bindungen ein:

```
x ↦ 5
f ↦ fn y => 5+y
g ↦ fn x => 2 * (fn y => 5+y) x
```

Die Darstellung von Prozeduren durch geschlossene Abstraktionen erklärt, warum die Redeklaration von Bezeichnern keine Auswirkung auf bereits berechnete Prozeduren hat. Wenn wir beispielsweise den obigen Deklarationen die Deklaration

```
val x = 88
```

nachfolgen lassen, bleiben die Bindungen der Bezeichner `f` und `g` unverändert.

Hier ist ein Beispiel für ein Auswertungsprotokoll, in dem Prozeduren durch Abstraktionen dargestellt werden:

```
(fn x => fn y => x+y) 2 3 → (fn y => 2+y) 3
                          → 2+3
                          → 5
```

Bei der Anwendung einer Abstraktion werden die benutzenden Auftreten der Argumentvariable durch den geschlossenen Ausdruck ersetzt, der das Argument darstellt. Im Zusammenhang mit Auswertungsprotokollen verzichten wir auf die Angabe von Typconstraints für Argumentvariablen, da diese bei der Ausführung keine Rolle spielen.

Wir kommen jetzt zu rekursiven Prozeduren. Diese stellen wir durch spezielle Ausdrücke der Form

`rec` *f* `=> fn` *x* `=>` *a*

dar, die wir als *rekursive Abstraktionen* bezeichnen. Dabei ist der Bezeichner *f* als lokaler Name für die beschriebene rekursive Prozedur zu verstehen. Das Auftreten von *f* nach dem Schlüsselwort `rec` ist definierend. Sein Gültigkeitsbereich ist die nachfolgende Abstraktion. Als Beispiel betrachten wir die Deklaration

```
fun f x y = if x=0 then y else y + f (x-1) y
```

Diese führt die Bindung

```
f ↦ rec f => fn x => fn y => if x=0 then y else y + f (x-1) y
```

ein. Bei der Anwendung einer rekursiven Abstraktion werden die benutzenden Auftreten des lokalen Prozedurnamens durch die rekursive Abstraktion ersetzt. Als Beispiel betrachten wir das Auswertungsprotokoll für den Ausdruck `f 1 13`. Damit wir das Protokoll übersichtlich darstellen können, verwenden wir ϕ als Abkürzung für die rekursive Abstraktion, die die Prozedur `f` darstellt.

```
f 1 13 → φ 1 13
       → (fn y => if 1=0 then y else y + φ (1-1) y) 13
       → if 1=0 then 13 else 13 + φ (1-1) 13
       → if false then 13 else 13 + φ (1-1) 13
       → 13 + φ (1-1) 13
       → 13 + φ 0 13
       → 13 + (fn y => if 0=0 then y else y + φ (0-1) y) 13
       → 13 + (if 0=0 then 13 else 13 + φ (0-1) 13)
       → 13 + (if true then 13 else 13 + φ (0-1) 13)
       → 13 + 13
       → 26
```

Übungsaufgaben

Aufgabe 3.1 (Höherstufige Prozeduren)

1. Deklarieren Sie eine Prozedur

   ```
   prod : (int -> int) * int -> int
   ```

   die die folgenden Gleichungen erfüllt:

   ```
   prod(f,0) = 1
   prod(f,n) = f n * prod(f,n-1)    falls n > 0
   ```

   Tun Sie dies mit einer Prozedurdeklaration.

2. Deklarieren Sie `prod` mit einer rekursiven Wertdeklaration.

3. Deklarieren Sie eine kaskadierte Variante von `prod` mit einer Prozedurdeklaration.

4. Deklarieren Sie eine kaskadierte Variante von `prod` mit einer rekursiven Wertdeklaration.

Aufgabe 3.2 (Until) Schreiben Sie eine Prozedur

```
until : (int -> bool) -> int
```

die zu einer Prozedur p das kleinste $n \in \mathbb{N}$ berechnet, für das p den Wert `true` liefert. Schreiben Sie `until` mithilfe eines Let-Ausdrucks, der eine Hilfsprozedur `until' : int -> int` deklariert.

Geben Sie mithilfe von `until` eine Abstraktion an, die eine Prozedur `int -> int` liefert, die zu $x \in \mathbb{N}$ das größte $n \in \mathbb{N}$ mit $n^2 \leq x$ berechnet.

Aufgabe 3.3 (Kartesische und kaskadierte Prozeduren) Deklarieren Sie zwei Prozeduren

```
kas : ('a * 'b -> 'c) -> 'a -> 'b -> 'c

kar : ('a -> 'b -> 'c) -> 'a * 'b -> 'c
```

für die gilt:

```
kar (kas op+) (3,4)
7 : int

kas (kar (kas op+)) 3 4
7 : int
```

Dabei soll `kas` zu einer kartesischen Prozedur eine gleichwertige kaskadierte Prozedur und `kar` zu einer kaskadierten Prozedur eine gleichwertige kartesische Prozedur liefern.

Aufgabe 3.4 (Monomorphe Typsynthese) Geben Sie Wertdeklarationen an, die die folgenden statischen Bindungen einführen:

```
val a : int * unit * bool
val b : unit * (int * unit) * (real * unit)
val c : int -> int
val d : int * bool -> int
val e : int -> real
val f : int -> real -> real
val g : (int -> int) -> bool
```

Dabei dürfen Sie die folgenden Dinge *nicht* verwenden:

- Typconstraints für Argumentvariablen.
- Prozeduranwendungen.
- Operationen (zum Beispiel `+`, `-`, `<` und `=`).
- Standardstrukturen.

Hinweis: Für einige der Bindungen ist die Verwendung eines Konditionals essentiell. Die Typregel für das Konditional verlangt, dass die Konsequenz und die Alternative den gleichen Typ haben. Für einige Bindungen benötigen Sie zusätzlich Let-Ausdrücke

`let val x =` a_1 `in` a_2 `end`

bei denen `x` nicht in a_2 vorkommt.

Aufgabe 3.5 (Polymorphe Typsynthese) Geben Sie für jedes der folgenden Typschemata einen polymorphen Ausdruck an, der genau die durch das Typschema beschriebenen Typen hat. Die Ausdrücke dürfen keine Typconstraints und keine freien Bezeichnerauftreten enthalten.

1. `∀ 'a ('a -> 'a)`
2. `∀ ''a (''a -> ''a)`
3. `∀ ''a 'b (''a -> 'b -> 'b * ''a)`
4. `∀ 'a 'b 'c (('a -> 'b -> 'c) -> 'a -> 'b -> 'c)`

Denken Sie daran, dass Typvariablen mit zwei Hochkommas nur mit Typen mit Gleichheit instanziiert werden können. Für einige Beispiele benötigen Sie den Gleicheitstest.

Aufgabe 3.6 (Bedeutungsgleiche Ausdrücke) Gegeben sei der folgende Ausdruck:

```
fn x => fn y => (fn x => (fn y => (fn x => y) x) y) x
```

1. Geben Sie einen möglichst einfachen bedeutungsgleichen Ausdruck an.
2. Geben Sie ein Typschema an, dass alle Typen des Ausdrucks beschreibt.

Aufgabe 3.7 (Lexikalische Bindungen) Betrachten Sie den Ausdruck

```
(fn x => (fn y => (fn x => y) x) y) x
```

1. Stellen Sie die lexikalischen Bindungen des Ausdrucks durch Überstreichen und Indizieren von Bezeichnerauftreten dar.
2. Welche Bezeichner kommen in dem Ausdruck frei vor?
3. Geben Sie einen bedeutungsgleichen bereinigten Ausdruck an.

Aufgabe 3.8 (Lexikalische Bindungen) Stellen Sie die lexikalischen Bindungen des folgenden Ausdrucks durch Überstreichen und Indizieren von Bezeichnerauftreten dar.

```
let
    val f = fn x => fn x => f x y
    val x = 2*x
    val x =(x,y,f)
    val rec g = fn n => if n<2 then 1 else n*g(n-1)
    fun f f = f x
in
    fn x => f x
end
```

Welche Bezeichner kommen in dem Ausdruck frei vor?