Kapitel 11

Semantik

In Kapitel 3 haben wir die programmiersprachlichen Aspekte einer Teilsprache von Standard ML betrachtet. Jetzt vertiefen wir diese Betrachtung und entwickeln eine präzise Definition für eine vereinfachte Variante dieser Teilsprache.

Die präzise Definition von Programmiersprachen ist keine ganz leichte Aufgabe. Wir sind allerdings hervorragend vorbereitet, da wir viele der benötigten Konzepte schon als programmiersprachliche Konzepte kennengelernt haben.

Man unterscheidet zwischen syntaktischen und semantischen Aspekten von Programmiersprachen. Die **Syntax** einer Programmiersprache regelt, wie man Programme darstellt. Die **Semantik** einer Programmiersprache regelt, welche Programme zulässig sind und welche Ergebnisse die Ausführung von zulässigen Programmen zu liefern hat.

Abbildung 11.1 zeigt die Zusammenhänge zwischen Syntax und Semantik. Im Zentrum steht die **abstrakte Syntax**, die die erforderlichen **syntaktischen Objekte** als mathematische Objekte definiert. Beispiele für syntaktische Objekte sind Ausdrücke, Typen und Deklarationen. Die **konkrete Syntax** regelt, wie die Objekte der abstrakten Syntax textuell, also als Folgen von Zeichen, repräsentiert werden. Die **statische Semantik** definiert Konsistenzbedingungen für die syntaktischen Objekte. Diese Bedingungen betreffen die Wohlgetyptheit und die Bindung von Bezeichnern. Die **dynamische Semantik** definiert, was bei der Ausführung von syntaktischen Objekten passieren soll.

In Kapitel 3 haben wir die drei Analysephasen eines Interpreters kennengelernt. Die lexikalische und syntaktische Analyse sind für die Übersetzung der konkreten Syntax in die abstrakte Syntax zuständig. Die sematische Analyse überprüft die Einhaltung der durch die statische Semantik auferlegten Konsistenzbedingungen.

Man unterscheidet zwischen statischen und dynamischen Aspekten von Programmiersprachen. Die die Analysephasen betreffenden Aspekte bezeichnet man

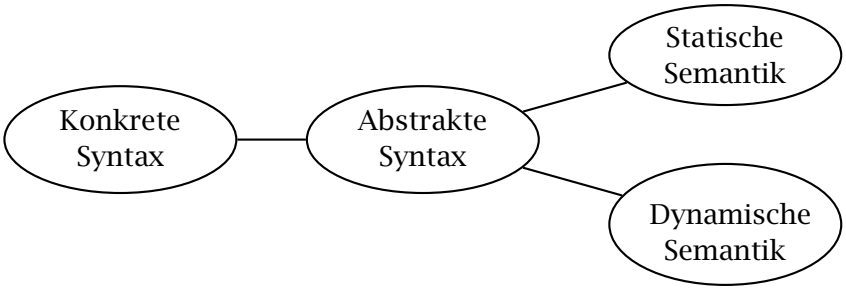


Abbildung 11.1: Syntax und Semantik

als **statisch**. Die die Ausführung betreffenden Aspekte bezeichnet man als **dynamisch**.

Idealisierte Programmiersprache F

Richtige Programmiersprachen sind sehr komplexe Artefakte. In diesem Kapitel betrachten wir eine kleine, idealisierte Programmiersprache, die wir F nennen. F ist eine monomorph typisierte Teilsprache von Standard ML, die trotz ihrer Einfachheit wesentliche programmiersprachliche Konzepte beinhaltet.

In diesem Kapitel entwickeln wir mathematische Definitionen für die abstrakte Syntax sowie die statische und dynamische Semantik von F. Aus den Definitionen der statischen und dynamischen Semantik leiten wir Realisierungen für die entsprechenden Verarbeitungsphasen eines Interpreters ab. Dabei legen wir keinen Wert auf Effizienz.

11.1 Abstrakte Syntax

Abbildung 11.2 definiert die abstrakte Syntax von F mittels einer schematischen Darstellung, die man als **abstrakte Grammatik** bezeichnet. Die Grammatik definiert fünf Mengen, deren Elemente als Konstanten, Bezeichner, Operatoren, Typen und Ausdrücke bezeichnet werden. Diese Mengen können auch mithilfe der Typdeklarationen in Abbildung 11.3 beschrieben werden. Mathematisch gesehen unterscheiden sich die durch die Grammatik und die Typdeklarationen definierten Mengen nur dadurch, dass die Grammatik Bezeichner als natürliche Zahlen modelliert, während die Typdeklarationen Bezeichner als Zeichenreihen realisieren. Wir erlauben uns diese kleine Abweichung, da es für die Belange der Semantik keine Rolle spielt, welche unendliche Menge wir für die Bezeichner wählen.

In Kapitel 5 haben wir die Werte von Konstruktortypen als mathematische Objekte definiert. Die Definitionen der Grammatik sind analog zu den Typdeklara-

c	$\in$	$Con = \texttt{false} \mid \texttt{true} \mid \mathbb{Z}$	**Konstanten**
x	$\in$	$Id = \mathbb{N}$	**Bezeichner**
o	$\in$	$Opr = + \mid - \mid * \mid \leq$	**Operatoren**
t	$\in$	$Ty = \texttt{bool} \mid \texttt{int} \mid t_1 \to t_2$	**Typen**
e	$\in$	$Exp =$	**Ausdrücke**
		c	**Konstante**
		$\mid x$	**Bezeichner**
		$\mid e_1\ o\ e_2$	**Operatoranwendung**
		$\mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$	**Konditional**
		$\mid \texttt{fn}\ x\texttt{:}\ t\ \texttt{=>}\ e$	**Abstraktion**
		$\mid e_1\ e_2$	**Prozeduranwendung**

Abbildung 11.2: Abstrakte Syntax von F (Abstrakte Grammatik)

```
datatype con = False | True | IC of int

type     id  = string

datatype opr = Add | Sub | Mul | Leq

datatype ty  = Bool | Int | Arrow of ty * ty

datatype exp = Con of con
             | Id  of id
             | Op  of exp * opr * exp
             | If  of exp * exp * exp
             | Abs of id  * ty  * exp
             | App of exp * exp
```

Abbildung 11.3: Abstrakte Syntax von F (Typdeklarationen)

tionen in Abbildung 11.3 zu verstehen, wobei wir idealisierend annehmen, dass die Werte von `int` die ganzen Zahlen sind. Die Menge der Konstanten ist also wie folgt definiert:

$$Con = \{\langle 1\rangle\} \cup \{\langle 2\rangle\} \cup \{\, \langle 3, z\rangle \mid z \in \mathbb{Z} \,\}$$

Die mathematische Essenz der Grammatik ist wie gesagt die Definition der Mengen *Con*, *Id*, *Opr*, *Ty* und *Exp*. Zusätzlich legt die Grammatik Notationen für die Elemente dieser Mengen fest. Diese Notationen sind unbedingt erforderlich, damit wir syntaktische Objekte in lesbarer Form beschreiben können. Beispielsweise können wie den Ausdruck

$$\begin{aligned}&\langle 4,\ \langle\langle 3, \langle\langle 2, 1\rangle, \langle 4\rangle, \langle 1, \langle 3, 1\rangle\rangle\rangle\rangle,\\ &\qquad \langle 1, \langle 3, 1\rangle\rangle,\\ &\qquad \langle 3, \langle\langle 2, 1\rangle, \langle 3\rangle, \langle 6, \langle 2, 2\rangle, \langle 3, \langle\langle 2, 1\rangle, \langle 2\rangle, \langle 1, \langle 3, 1\rangle\rangle\rangle\rangle\rangle\rangle\rangle\rangle\rangle\rangle\end{aligned}$$

mit den durch die Grammatik festgelegten Notationen lesbar wie folgt beschreiben:

$$\texttt{if}\ x1 \leq 1\ \texttt{then}\ 1\ \texttt{else}\ x1 * x2\,(x1 - 1)$$

Die Typdeklarationen in Abbildung 11.3 legen alternative Notationen für die syntaktischen Objekte fest. Damit kann der obige Ausdruck wie folgt beschrieben werden:

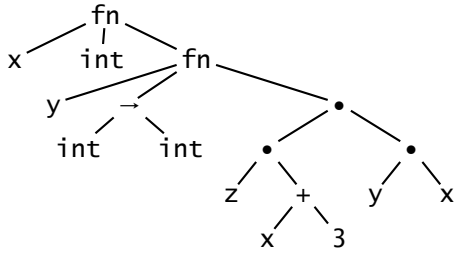
```
If(Op(Id "x1", Leq, Con(IC 1)),
   Con(IC 1),
   Op(Id "x1", Mul, App(Id "x2", Op(Id "x1", Sub, Con(IC 1)))))
```

Die durch die Grammatik festgelegten mathematischen Notationen sind sehr kompakt. Das wird mit diversen notationalen Tricks erkauft, die auf die Intelligenz des menschlichen Lesers vertrauen. Zum Beispiel wird auf einige der eigentlich notwendige Konstruktoren verzichtet (die Typdeklarationen führen alle Konstruktoren ein). Die fehlenden Konstruktoren muss der menschliche Leser aus dem Kontext ableiten und an den richtigen Stellen einfügen.

Die Grammatik legt nicht fest, ob die Beschreibung

$$1 + 2 + 3$$

als der Ausdruck $(1 + 2) + 3$ oder der Ausdruck $1 + (2 + 3)$ zu interpretieren ist. Dieses Problem kann man dadurch lösen, dass man zusätzliche Interpretationsregeln angibt. Wir verzichten hier auf solche Regeln und sorgen stattdessen mithilfe von Klammern dafür, dass die verwendeten Beschreibungen eindeutig interpretiert werden können.




```
fn x:int => fn y:int→int => z (x+3) (y x)
```

Abbildung 11.4: Baumdarstellung eines Ausdrucks

Die in der Grammatik verwendeten Buchstaben *c*, *x*, *o*, *t* und *e* dienen als notationale Variablen und werden als **Metavariablen** bezeichnet. Im Kontext der Grammatik bezeichnet eine Metavariable stets ein beliebiges Element der ihr durch die Grammatik zugeordneten Menge. Beispielsweise bezeichnet *t* im Kontext der Grammatik stets einen Typ.

Wir haben jetzt zum ersten Mal präzise gesagt, um welche mathematischen Objekte es sich bei Ausdrücken und Typen handelt. Die dazu notwendigen Konzepte haben wir bereits im Zusammenhang mit Konstruktortypen kennengelernt.

Die Formalisierung von Ausdrücken als Tupel macht die Baumstruktur von Ausdrücken explizit. Es ist oft nützlich, die Baumstruktur eines Ausdrucks grafisch darzustellen. Abbildung 11.4 zeigt ein Beispiel.

Die durch die Grammatik festgelegten Notationen suggerieren, wie wir F als Teilsprache von Standard ML auffassen können. Wir werden die statische und dynamische Semantik von F genau gemäß dieser Interpretation definieren. Das hat den Vorteil, dass wir F nicht weiter erklären müssen und eine klare Vorstellung davon haben, was wir definieren wollen. Außerdem können wir für Beispiele die konkrete Syntax von Standard ML für F benutzen.

Wir werden die dynamische Semantik von F ohne Größenbeschränkungen für Zahlen und ohne Speicherbeschränkungen definieren. Auch die Definition von Standard ML verzichtet aus guten Gründen auf diese Beschränkungen. Es ist durchaus möglich, Standard ML ohne Größenbeschränkungen für ganze Zahlen zu implementieren (siehe Abschnitt 10.2). Dagegen sind Speicherbeschränkungen von prinzipieller Natur. Sie sind abhängig von der Hardware des Computers, auf dem ein Programm ausgeführt wird, und der Art und Weise, wie ein Programmiersystem den zur Verfügung stehenden Speicher verwaltet.

11.2 Statische Semantik

Die statische Semantik definiert Konsistenzbedingungen für die syntaktischen Objekte. Diese Bedingungen betreffen die Wohlgetyptheit und die Bindung von Bezeichnern.

Im Kontext der statischen Semantik werden Bezeichner an Typen gebunden. Eine Menge solcher Bindungen bezeichnet man als eine Umgebung. Genauer verstehen wir unter einer **Typumgebung** (englisch type environment) eine endliche Funktion, die Bezeichner auf Typen abbildet. Wir verwenden die folgenden Schreibweisen:

$$T \in TE = Id \xrightarrow{fin} Ty$$

Wir definieren die statische Semantik von F als eine Menge

$$SS \subseteq TE \times Exp \times Ty$$

und schreiben

$$T \vdash e \Longrightarrow t$$

für

$$(T,\ e,\ t) \in SS$$

Dabei soll $T \vdash e \Longrightarrow t$ genau dann gelten, wenn der Ausdruck e in der Typumgebung T wohlgetypt ist und den Typ t hat. Die Typumgebung T ist dafür zuständig, den in e frei auftretenden Bezeichnern Typen zuzuordnen.

Die Menge SS definieren wir rekursiv durch die Inferenzregeln in Abbildung 11.5. Die Notation $T + \{x \mapsto t\}$ wurde in Abschnitt 2.5 definiert. Für jede der sechs Ausdrucksvarianten gibt es mindestens eine Regel. Die in diesen Regeln vorkommenden Metavariablen (zum Beispiel T oder c) dürfen dabei nur an Objekte der jeweiligen Klasse gebunden werden. Die Definition der statischen Semantik durch die Inferenzregeln ist kompakt und modular (jede Ausdrucksvariante wird für sich behandelt). Machen Sie sich klar, dass die Regeln neben der Typanalyse auch die notwendige Bindungsanalyse formulieren.

Die folgende Proposition stellt fest, dass ein Ausdruck in einer Typumgebung höchstens einen Typ hat.

Proposition 11.2.1 (Determinismus) *Sei $T \vdash e \Longrightarrow t$ und $T \vdash e \Longrightarrow t'$. Dann gilt $t = t'$.*

$$\frac{}{T \vdash \texttt{false} \Longrightarrow \texttt{bool}} \qquad \frac{}{T \vdash \texttt{true} \Longrightarrow \texttt{bool}}$$

$$\frac{c \in \mathbb{Z}}{T \vdash c \Longrightarrow \texttt{int}} \qquad \frac{T(x) = t}{T \vdash x \Longrightarrow t}$$

$$\frac{o \in \{+, -, *\} \quad T \vdash e_1 \Longrightarrow \texttt{int} \quad T \vdash e_2 \Longrightarrow \texttt{int}}{T \vdash e_1\ o\ e_2 \Longrightarrow \texttt{int}}$$

$$\frac{T \vdash e_1 \Longrightarrow \texttt{int} \quad T \vdash e_2 \Longrightarrow \texttt{int}}{T \vdash e_1 \leq e_2 \Longrightarrow \texttt{bool}}$$

$$\frac{T \vdash e_1 \Longrightarrow \texttt{bool} \quad T \vdash e_2 \Longrightarrow t \quad T \vdash e_3 \Longrightarrow t}{T \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Longrightarrow t}$$

$$\frac{T + \{x \mapsto t\} \vdash e \Longrightarrow t'}{T \vdash \texttt{fn}\ x\texttt{:}\ t\ \texttt{=>}\ e \Longrightarrow t \to t'}$$

$$\frac{T \vdash e_1 \Longrightarrow t' \to t \quad T \vdash e_2 \Longrightarrow t'}{T \vdash e_1\ e_2 \Longrightarrow t}$$

Abbildung 11.5: Statische Semantik von F

```
type 'a env
exception Unbound of id
val empty  : 'a env
val insert : id * 'a * 'a env → 'a env
val lookup : id * 'a env → 'a  (* Unbound *)
```

Abbildung 11.6: Signatur für Umgebungen

Ein Ausdruck e heißt **zulässig** genau dann, wenn es einen Typ t gibt mit $\varnothing \vdash e \Longrightarrow t$.

Proposition 11.2.2 *Jeder zulässige Ausdruck ist geschlossen.*

Man kann die Regeln der statischen Semantik direkt in eine rekursive Prozedur `elab` umformulieren, die zu einer Typumgebung T und zu einem Ausdruck e feststellt, ob ein Typ t existiert, so dass $T \vdash e \Longrightarrow t$ gilt. Wenn ein Typ existiert, wird er als Ergebnis geliefert. Wenn kein Typ existiert, wird eine Ausnahme geworfen, die beschreibt, welcher statische Fehler vorliegt.

Wir nehmen an, dass Umgebungen gemäß der Signatur in Abbildung 11.6 realisiert sind. Abbildung 11.7 zeigt die Realisierung der Prozedur `elab`. Diese ist mit einer Fallunterscheidung nach den sechs Ausdrucksvarianten realisiert. Jeder Fall realisiert die entsprechenden Inferenzregeln. Die rekursiven Aufrufe werden durch die Prämissen der Regeln beschrieben. Die Rekursion terminiert, da die Ausdrücke in den Prämissen einer Regel stets kleiner sind als der Ausdruck in der Konklusion der Regel.

11.3 Dynamische Semantik

Die dynamische Semantik legt fest, ob und mit welchem Wert die Auswertung eines Ausdrucks terminiert.

F hat zwei Arten von Werten, ganze Zahlen und Prozeduren. Die Booleschen Werte stellen wir, wie in Abschnitt 3.1 besprochen, durch die Zahlen 0 und 1 dar.

In Kapitel 1 haben wir gelernt, das man eine Prozedur durch ein Paar darstellen kann, das aus einer Prozedurdeklaration und einer Umgebung besteht, die die freien Variablen der Deklaration an Werte bindet. Statt mit Prozedurdeklarationen arbeiten wir in F mit Abstraktionen. Die für Prozeduren benötigten *Wertumgebungen* spielen in der dynamischen Semantik eine ähnliche Rolle wie Typumgebungen in der statischen Semantik.

```
exception Error of string

fun elabCon True   = Bool
  | elabCon False  = Bool
  | elabCon (IC _) = Int

fun elabOp Int Add Int = Int
  | elabOp Int Sub Int = Int
  | elabOp Int Mul Int = Int
  | elabOp Int Leq Int = Bool
  | elabOp _   _   _   = raise Error "ill-typed operation"

fun elab T (Con c)         = elabCon c

  | elab T (Id s)          = lookup(s,T)

  | elab T (Op(e1,opr,e2)) = elabOp (elab T e1) opr (elab T e2)

  | elab T (If(e1,e2,e3))  = let val t1 = elab T e1
                                 val t2 = elab T e2
                                 val t3 = elab T e3
                             in  if t1=Bool andalso t2=t3 then t2
                                 else raise
                                    Error "ill-typed conditional"
                             end

  | elab T (App(e1,e2))    = let val t1 = elab T e1
                                 val t2 = elab T e2
                                 val s  = "ill-typed application"
                             in  case t1 of
                                Arrow(t,t') => if t=t2 then t'
                                               else raise Error s
                                | _  => raise Error s
                             end

  | elab T (Abs(s,t,e))    = Arrow(t, elab (insert(s,t,T)) e)

val elab  : ty env → exp → ty  (* Error, Unbound *)
```

Abbildung 11.7: Die Prozedur `elab`

$$
\begin{array}{lcll}
v \in Val & = & \mathbb{Z} \cup Pro & \textbf{Werte} \\
 & Pro & = & Id \times Exp \times VE \quad \textbf{Prozeduren} \\
V \in VE & = & Id \xrightarrow{fin} Val & \textbf{Wertumgebungen}
\end{array}
$$

Abbildung 11.8: Werte, Prozeduren und Wertumgebungen für F

Wir stellen die Prozedur, die durch die Auswertung einer Abstraktion

`fn` x`:` t `=>` e

erhalten wird, durch ein Tripel

$$(x,\ e,\ V)$$

dar, dessen dritte Komponente V eine Wertumgebung ist, die den in der Abstraktion frei auftretenden Bezeichnern Werte zuordnet. Mit dieser Darstellung liefert die Auswertung des Ausdrucks

```
(fn x:int => fn y:int => x+y) 7
```

die Prozedur

$$(y,\ x + y,\ \{x \mapsto 7\})$$

Beachten Sie, dass die Typen der Argumentvariablen nicht in die Darstellung der Prozedur aufgenommen werden. Das liegt daran, dass sie bei der Anwendung der Prozedur nicht benötigt werden.

Die **Werte** (englisch values), **Prozeduren** (englisch procedures) und **Wertumgebungen** (englisch value environments) für die dynamische Semantik von F definieren wir verschränkt rekursiv wie in Abbildung 11.8 gezeigt.

Wir definieren die dynamische Semantik von F als eine Menge

$$DS \subseteq VE \times Exp \times Val$$

und schreiben

$$V \vdash e \Longrightarrow v$$

für

$$(V,\ e,\ v) \in DS$$

Dabei soll $V \vdash e \Longrightarrow v$ genau dann gelten, wenn die Evaluation des Ausdrucks e in der Wertumgebung V terminiert und den Wert v liefert. Die Wertumgebung V ist dafür zuständig, den in e frei auftretenden Bezeichnern Werte zuzuordnen.

Die Menge *DS* definieren wir rekursiv durch die Inferenzregeln in Abbildung 11.9. Die Regel für Abstraktionen liefert Prozeduren, die die gesamte Umgebung mit sich tragen, die bei der Auswertung der Abstraktion vorlag. Das ist die einfachste Lösung. Alternativ könnten wir einer Prozedur nur den Teil der Umgebung mitgeben, der die in der Abstraktion frei auftretenden Bezeichner bindet.

Die folgende Proposition besagt, dass das Ergebnis der Auswertung eines Ausdrucks eindeutig bestimmt ist:

Proposition 11.3.1 (Determinismus) *Sei $V \vdash e \Longrightarrow v$ und $V \vdash e \Longrightarrow v'$. Dann gilt $v = v'$.*

Der folgende Satz stellt fest, dass es in F weder Divergenz noch Laufzeitfehler gibt.

Satz 11.3.2 (Auswertbarkeit) *Für jeden zulässigen Ausdruck e existiert genau ein Wert v mit $\emptyset \vdash e \Longrightarrow v$.*

Der Auswertbarkeitssatz ist keineswegs einfach zu beweisen. Er bestätigt unsere Intuition, dass man ohne Rekursion keinen zulässigen Ausdruck angeben kann, dessen Auswertung divergiert.

Satz 11.3.3 (Typerhaltung)

1. *Sei $\emptyset \vdash e \Longrightarrow \texttt{int}$ und $\emptyset \vdash e \Longrightarrow v$. Dann $v \in \mathbb{Z}$.*
2. *Sei $\emptyset \vdash e \Longrightarrow \texttt{bool}$ und $\emptyset \vdash e \Longrightarrow v$. Dann $v \in \{0, 1\}$.*

Zulässige Ausdrücke der Typen `int` und `bool` evaluieren also immer zu Werten dieser Typen.

Der Auswertbarkeitssatz und der Typerhaltungssatz liefern wichtige Sicherheitsgarantien für zulässige Ausdrücke.

Die Werte eines Typs t können wir wie folgt definieren:

$$\mathit{Value}(t) = \{v \in \mathit{Val} \mid \exists e\colon\ \emptyset \vdash e \Longrightarrow t\ \wedge\ \emptyset \vdash e \Longrightarrow v\}$$

Offensichtlich gibt es in *Pro* „unechte“ Prozeduren, die zu keinem Typ gehören, zum Beispiel $(x,\ y\ x,\ \{y \mapsto 1\})$.

$$\frac{}{V \vdash \texttt{false} \Longrightarrow 0} \qquad \frac{}{V \vdash \texttt{true} \Longrightarrow 1} \qquad \frac{c \in \mathbb{Z}}{V \vdash c \Longrightarrow c}$$

$$\frac{V(x) = v}{V \vdash x \Longrightarrow v}$$

$$\frac{V \vdash e_1 \Longrightarrow v_1 \quad V \vdash e_2 \Longrightarrow v_2 \quad v_1, v_2 \in \mathbb{Z} \quad v = v_1 + v_2}{V \vdash e_1 + e_2 \Longrightarrow v}$$

$$\frac{V \vdash e_1 \Longrightarrow v_1 \quad V \vdash e_2 \Longrightarrow v_2 \quad v_1, v_2 \in \mathbb{Z} \quad v = v_1 - v_2}{V \vdash e_1 - e_2 \Longrightarrow v}$$

$$\frac{V \vdash e_1 \Longrightarrow v_1 \quad V \vdash e_2 \Longrightarrow v_2 \quad v_1, v_2 \in \mathbb{Z} \quad v = v_1 \cdot v_2}{V \vdash e_1 * e_2 \Longrightarrow v}$$

$$\frac{V \vdash e_1 \Longrightarrow v_1 \quad V \vdash e_2 \Longrightarrow v_2 \quad v_1, v_2 \in \mathbb{Z} \quad v = \text{if } v_1 \leq v_2 \text{ then } 1 \text{ else } 0}{V \vdash e_1 \leq e_2 \Longrightarrow v}$$

$$\frac{V \vdash e_1 \Longrightarrow 1 \qquad V \vdash e_2 \Longrightarrow v}{V \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Longrightarrow v}$$

$$\frac{V \vdash e_1 \Longrightarrow 0 \qquad V \vdash e_3 \Longrightarrow v}{V \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Longrightarrow v}$$

$$\frac{}{V \vdash \texttt{fn}\ x\texttt{:}\ t\ \texttt{=>}\ e \Longrightarrow (x, e, V)}$$

$$\frac{V \vdash e_1 \Longrightarrow (x, e, V') \quad V \vdash e_2 \Longrightarrow v_2 \quad V' + \{x \mapsto v_2\} \vdash e \Longrightarrow v}{V \vdash e_1\ e_2 \Longrightarrow v}$$

Abbildung 11.9: Dynamische Semantik von F

Beachten Sie, dass die Inferenzregeln in Abbildung 11.9 die Ordnung, in der Teilausdrücke ausgewertet werden sollen, nur teilweise festlegen. Beispielsweise lässt die Regel für Addition

$$\frac{V \vdash e_1 \Longrightarrow v_1 \quad V \vdash e_2 \Longrightarrow v_2 \quad v_1, v_2 \in \mathbb{Z} \quad v = v_1 + v_2}{V \vdash e_1 + e_2 \Longrightarrow v}$$

offen, ob e_1 oder e_2 zuerst ausgewertet werden soll. Eine weitergehende Festlegung ist unnötig, da sie an der zu definierenden Menge *DS* nichts ändern würde.

Die Regeln der dynamischen Semantik kann man wie bei der statischen Semantik zu einer rekursiven Prozedur

```
eval : value env → exp → value
```

umformulieren, die einen Ausdruck e in einer Wertumgebung V evaluiert und seinen Wert als Ergebnis liefert. Abbildung 11.10 zeigt die Realisierung der Prozedur `eval`. Der Auswertbarkeitssatz 11.3.2 garantiert, dass `eval` für zulässige Ausdrücke immer regulär terminiert (also ohne eine Ausnahme zu werfen). Der Typerhaltungssatz 11.3.3 garantiert, dass `eval` für zulässige Ausdrücke des Typs `int` [`bool`] immer ein Ergebnis der Form `IV` *n* [`IV 0` oder `IV 1`] liefert.

Die dynamische Semantik ist unabhängig von der statischen Semantik definiert. Die definierenden Inferenzregeln und die Prozedur `eval` ignorieren die in den Abstraktionen vermerkten Typen für die Argumentvariablen völlig. Für die dynamische Semantik spielt es also keine Rolle, welche Typen man für die Argumentvariablen angibt. Man hat also die Möglichkeit, die in einem Ausdruck angegebenen Typen vor der Ausführung vollständig zu löschen. Von dieser Möglichkeit machen Programmiersysteme für SML aus Effizienzgründen in der Tat Gebrauch.

11.4 Let-Ausdrücke und Paare

Einen Let-Ausdruck

`let val` x`:` t `=` e_1 `in` e_2 `end`

kann man auch mit einer Abstraktion und einer Applikation ausdrücken:

(`fn` x`:` t `=>` e_2) e_1

Das gibt uns die Möglichkeit, die konkrete Syntax von F mit Let-Ausdrücken auszustatten, ohne die abstrakte Syntax zu erweitern. Die syntaktische Analyse übernimmt dann die Übersetzung von Let-Ausdrücken in die entsprechenden Abstraktionen und Applikationen. Man sagt, dass Let-Ausdrücke als **abgeleitete Form** zu F hinzugefügt werden können.

```
datatype value = IV    of int
               | Proc  of id * exp * value env

fun evalCon False  = IV 0
  | evalCon True   = IV 1
  | evalCon (IC x) = IV x

fun evalOp (IV x1) Add (IV x2) = IV(x1+x2)
  | evalOp (IV x1) Sub (IV x2) = IV(x1-x2)
  | evalOp (IV x1) Mul (IV x2) = IV(x1*x2)
  | evalOp (IV x1) Leq (IV x2) = IV(if x1<=x2 then 1 else 0)
  | evalOp  _      _   _       = raise Error "type error"

fun eval V (Con c)         = evalCon c

  | eval V (Id  s)         = lookup(s,V)

  | eval V (Op(e1,opr,e2)) = evalOp (eval V e1) opr (eval V e2)

  | eval V (If(e1,e2,e3))  = (case eval V e1 of
                                 IV 1 => eval V e2
                               | IV 0 => eval V e3
                               |  _   => raise Error "type error")

  | eval V (Abs(s,_,e))    = Proc(s,e,V)

  | eval V (App(e1,e2))    = (case (eval V e1, eval V e2) of
                                 (Proc(s,e,V'), v) =>
                                   eval (insert(s,v,V')) e
                               | _ => raise Error "type error")

val eval  : value env → exp → value  (* Error, Unbound *)
```

Abbildung 11.10: Die Prozedur eval

Auch Let-Ausdrücke mit mehr als einer Deklaration können auf Abstraktion und Applikation zurückgeführt werden. Beispielsweise können wir den Let-Asudruck

`let` d_1 d_2 `in` e `end`

auf zwei ineinander geschachtelte Let-Ausdrücke mit je einer Deklaration zurückführen:

`let` d_1 `in let` d_2 `in` e `end end`

Das Hinzufügen von Let-Ausdrücken als abgeleitete Form hat den Nachteil, dass der Programmierer den Typ der deklarierten Variable explizit angeben muss. Wenn wir Let-Ausdrücke in die abstrakte Syntax aufnehmen, können wir wegen Proposition 11.2.1 auf die Angabe eines Typs verzichten. Hier sind die dann erforderlichen Regeln für die statische und dynamische Semantik:

$$\frac{T \vdash e_1 \Longrightarrow t_1 \qquad T + \{x \mapsto t_1\} \vdash e_2 \Longrightarrow t}{T \vdash \texttt{let val } x = e_1 \texttt{ in } e_2 \texttt{ end} \Longrightarrow t}$$

$$\frac{V \vdash e_1 \Longrightarrow v_1 \qquad V + \{x \mapsto v_1\} \vdash e_2 \Longrightarrow v}{V \vdash \texttt{let val } x = e_1 \texttt{ in } e_2 \texttt{ end} \Longrightarrow v}$$

Auch Paare des Typs `int*int` können auf die in F vorhandenen Sprachmittel zurückgeführt werden. Ein Paar des Typs `int*int` können wir durch eine Prozedur `bool→int` darstellen, die für `true` die erste und für `false` die zweite Komponente des Paars liefert. Hier sind Deklarationen für Prozeduren, die Paare konstruieren und zerlegen:

```
val pair = fn x:int => fn y:int => fn b:bool =>
              if b then x else y

val fst = fn p:bool→int => p true

val snd = fn p:bool→int => p false
```

Mit der gerade gezeigten Methode können wir **homogene Paare** des Typs $t * t$ darstellen. **Heterogene Paare** des Typs $t_1 * t_2$ mit $t_1 \neq t_2$ lassen sich dagegen nicht in F ausdrücken, da die für eine Darstellung erforderlichen Typen fehlen (da die Konsequenz und die Alternative eines Konditionals den gleichen Typ haben müssen).

11.5 Rekursive Prozeduren

Wir erweitern F jetzt um rekursive Prozeduren. Die erweiterte Sprache nennen wir FR. Für rekursive Prozeduren benötigen wir zusätzliche Syntax. Diese wählen

wir anders als in Standard ML. Wir erweitern die abstrakte Syntax von F um eine neue Ausdrucksform

$$e \in Exp \;=\; \cdots \mid \texttt{rec}\; x_1(x_2\texttt{:}\; t_2)\texttt{:}\; t_1\; \texttt{=>}\; e$$

die wir als **rekursive Abstraktion** bezeichnen. Dabei fungiert x_1 als Name für die beschriebene Prozedur und x_2 als Argumentvariable. Bei t_1 handelt es sich um den Argumentyp und bei t_2 um den Ergebnistyp der Prozedur. Die Bedeutung der neuen Ausdrucksvariante können wir in Standard ML wie folgt beschreiben:

```
let fun x1 (x2:t2): t1 = e in x1 end
```

Eine rekursive Prozedur, die die Fakultätsfunktion berechnet, können wir mit einer rekursiven Abstraktion wie folgt schreiben:

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

Wenn wir Standard ML um rekursive Abstraktionen erweitern würden, könnten wir Prozedurdeklarationen der Form

```
fun x1 (x2:t2): t1 = e
```

auf Wertdeklarationen der Form

```
val x1 = rec x1 (x2:t2): t1 => e
```

zurückführen.

Die statische Semantik erweitern wir um die Inferenzregel:

$$\frac{T + \{x_1 \mapsto t_2 \to t_1\} + \{x_2 \mapsto t_2\} \vdash e \Longrightarrow t_1}{T \vdash \texttt{rec}\; x_1(x_2\texttt{:}\; t_2)\texttt{:}\; t_1\; \texttt{=>}\; e \Longrightarrow t_2 \to t_1}$$

Um die dynamische Semantik um rekursive Prozeduren zu erweitern, müssen wir zuerst rekursive Prozeduren als Werte darstellen. Dazu wählen wir vierstellige Tupel:

$$Val = \mathbb{Z} \cup Pro \cup RPro$$

$$RPro = Id \times Id \times Exp \times VE$$

Die Bedeutung dieser Darstellung wird durch die Inferenzregel für rekursive Abstraktionen klar:

$$\frac{}{V \vdash \texttt{rec}\; x_1(x_2\texttt{:}\; t_2)\texttt{:}\; t_1\; \texttt{=>}\; e \Longrightarrow (x_1, x_2, e, V)}$$

Zusätzlich benötigen wir eine Inferenzregel für die Anwendung von rekursiven Prozeduren:

$$\frac{V \vdash e_1 \Longrightarrow v_1 \qquad v_1 = (x_1, x_2, e, V') \qquad V \vdash e_2 \Longrightarrow v_2 \qquad V' + \{x_1 \mapsto v_1\} + \{x_2 \mapsto v_2\} \vdash e \Longrightarrow v}{V \vdash e_1\; e_2 \Longrightarrow v}$$

Mit rekursiven Abstraktionen kann man zulässige Ausdrücke schreiben, deren Auswertung divergiert. Folglich ist der Auswertbarkeitssatz nicht auf FR übertragbar. Proposition 11.2.1 (Determinismus) und Satz 11.3.3 (Typerhaltung) gelten jedoch auch für FR.

Bei der Erweiterung von F um rekursive Prozeduren haben wir sehr von der modularen Struktur der Definition von F profitiert: Die existierenden Teile konnten ohne Überarbeitung mit den neuen Teilen kombiniert werden.

Um FR in Standard ML zu realisieren, erweitern wir die Deklaration von `exp` um einen Konstruktor für rekursive Abstraktionen.

```
datatype exp = ··· | Rec of id * id * ty * ty * exp
```

Die Erweiterung der Prozedur `elab` ergibt sich unmittelbar aus der Inferenzregel für die statische Semantik von rekursiven Abstraktionen. Für die Realisierung der dynamischen Semantik erweitern wir die Deklaration von `value` um einen Konstruktor für rekursive Prozeduren:

```
datatype value = ··· | RProc of id * id * exp * value env
```

Die Erweiterung der Prozedur `eval` ergibt sich aus den Inferenzregeln für die dynamische Semantik von rekursiven Abstraktionen.