5.3 Operationale Semantik

Das Grundprinzip der Auswertung bei funktionalen Programmiersprachen ist die Ersetzung „Gleiches durch Gleiches“ mittels der Anwendung von Funktionsgleichungen/Regeln von links nach rechts. Aus diesem Grund ist es wichtig festzulegen, wann eine Regel anwendbar ist. Da in Regeln auch Muster vorkommen können, verlangt dieser Begriff einige Vorbereitungen.

Definition 5.1 (Ausdruck)**.** *Ein **Ausdruck** ist eine Variable x oder eine Anwendung $(f\ e_1 \ldots e_n)$, wobei f eine Funktion oder ein Konstruktor ist und $e_1, \ldots, e_n$ wiederum Ausdrücke sind ($n \geq 0$).*

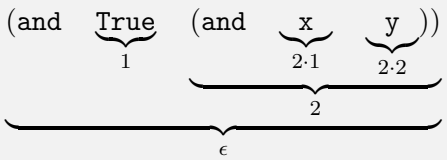
Wir ignorieren hier zunächst die Forderung nach Wohlgetyptheit, denn darauf gehen wir in einem späteren Kapitel ein.

Definition 5.2 (Position)**.** *Eine **Position** ist eine Liste natürlicher Zahlen, die einen Teilausdruck identifiziert.*

Definition 5.3 (Teilausdruck)**.** *Ein **Teilausdruck** $e|_p$ („e an der Stelle p“), wobei p eine Position im Ausdruck e ist, ist formal wie folgt definiert:*

$$e|_\epsilon = e \qquad (\epsilon\text{: leere Liste, Wurzelposition})$$
$$(f\ e_1 \ldots e_n)|_{i \cdot p} = e_i|_p$$

Beispiel 5.10 (Ausdruck und Positionen)**.**



Definition 5.4 (Ersetzung)**.** *Die **Ersetzung** eines Teilausdrucks in einem Ausdruck e an der Position p durch einen neuen Teilausdruck t wird durch $e[t]_p$ notiert. Formal ist dies wie folgt definiert:*

$$e[t]_\epsilon = t$$
$$(f\ e_1 \ldots e_n)[t]_{i \cdot p} = (f\ e_1 \ldots e_{i-1}\ e_i[t]_p\ e_{i+1} \ldots e_n)$$

Definition 5.5 (Substitution)**.** *Eine **Substitution** ist eine Ersetzung von Variablen durch Ausdrücke, wobei gleiche Variablen durch gleiche Ausdrücke ersetzt werden. Wir notieren Substitutionen in der Form*

$$\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$$

Die Formalisierung der Anwendung einer Substitution auf einen Ausdruck ist gegeben als:

$$\begin{aligned} \sigma(x_i) &= t_i && (i = 1, \ldots, k) \\ \sigma(x) &= x && \textit{falls } x \neq x_i \textit{ für alle } i = 1, \ldots, k \\ \sigma(f\, e_1 \ldots e_n) &= f\ \sigma(e_1) \ldots \sigma(e_n) \end{aligned}$$

Damit haben wir nun alle Begriffe bereitgestellt, um einen Ersetzungsschritt („ersetze Gleiches durch Gleiches“) zu definieren.

Definition 5.6 (Ersetzungsschritt, Redex)**.** *Ein **Ersetzungsschritt**, geschrieben*

$$e \to e'$$

*ist möglich, falls $l = r$ eine Funktionsgleichung ist, p eine Position in e und σ eine Substitution mit $\sigma(l) = e|_p$ und $e' = e[\sigma(r)]_p$. In diesem Fall heißt $e|_p$ auch **Redex** (reducible expression) von e.*

Eine funktionale Berechnung ist eine Folge von Ersetzungsschritten. Wenn der sich ergebende Ausdruck keine definierten Funktionen mehr enthält, ist diese erfolgreich.

Definition 5.7 (Erfolgreiche Berechnung)**.** *Eine **erfolgreiche Berechnung** eines Ausdrucks e ist eine endliche Folge*

$$e_1 \to e_2 \to \cdots \to e_n$$

wobei e_n keine definierten Funktionen enthält. In diesem Fall heißt e_n auch Wert von e_1:

$$e_n = value(e_1)$$

Bisher haben wir nur Funktionsregeln ohne Bedingungen betrachtet. Im allgemeinen Fall sind einige Erweiterungen einzubeziehen:

1. Regeln für `if-then-else`: Bedingte Ausdrücke können wir als Funktionsanwendung in mixfix-Schreibweise auffassen, die durch folgende Regeln definiert sind:

   ```
   if True  then x else y = x
   if False then x else y = y
   ```

2. Regeln für `case`: Dies entspricht einem Regelschema, ist also keine wirkliche Funktionsregel.

   `(case (C` $x_1 \ldots x_n$`) of { ...; C` $x_1 \ldots x_n$ `→`e`; ... }) =` e

3. Bedingte Regeln: Transformiere eine bedingte Gleichung der Form

l | b_1 = r_1
⋮
| b_k = r_k

in eine unbedingte Gleichung der Form

l = if b_1 then r_1 else
if b_2 then r_2 else
⋮
if b_k then r_k else error ...

4. Primitive Funktionen: Diese können wir so interpretieren, dass sie durch eine unendliche Menge von Gleichungen definiert sind, wie z.B.

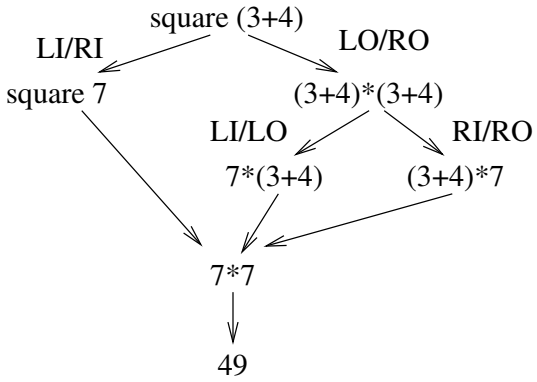
```
0 + 0 = 0          3 <  5 = True
⋮                  ⋮
3 + 4 = 7          3 == 5 = False
⋮                  ⋮
```

5.3.1 Auswertungsstrategien

Im Allgemeinen gibt es mehrere mögliche Berechnungen für einen gegebenen Ausdruck. Betrachten wir z. B. die Funktion

```
square x = x * x
```

Dann gibt es für den Ausdruck „`square (3 + 4)`“ folgende Berechnungen:



Eine **Auswertungsstrategie** legt für jeden Ausdruck den zu ersetzenden Redex fest. Wichtige Auswertungsstrategien sind beispielsweise

Innermost Der ersetzte Redex enthält keine anderen Redexe.

Leftmost Innermost (LI) Der ersetzte Redex ist der linkeste innerste.

Rightmost Innermost (RI) Der ersetzte Redex ist der rechteste innerste.

Outermost Der ersetzte Redex ist nicht Teil eines anderen Redex.

Leftmost Outermost (LO) Der ersetzte Redex ist der linkeste äußerste.

Rightmost Outermost (RO) Der ersetzte Redex ist der rechteste äußerste.

Funktionale Sprachen können anhand ihrer Auswertungsstrategie in strikte und nichtstrikte Sprachen unterschieden werden.

Strikte funktionale Sprachen (Lisp, Scheme, SML): Diese Sprachen verwenden die LI-Strategie, außer bei `if-then-else`- und `case`-Ausdrücken, denn diese werden outermost ausgewertet. Dies entspricht der Parameterübergabe call-by-value.

Nicht-strikte funktionale Sprachen (Miranda, Gofer, Haskell): Diese Sprachen verwenden die LO-Strategie, dies entspricht der Parameterübergabe call-by-name.

Im obigen Beispiel führen alle Strategien zum gleichen Ergebnis. Dies muss nicht immer so sein, allerdings gilt der folgende Satz:

Satz 5.1 (**Konfluenz/Eindeutigkeit der Werte**)**.** *Falls für einen Ausdruck e zwei Ableitungen*

$$e \to \cdots \to e_1$$

und

$$e \to \cdots \to e_2$$

existieren, wobei e_1, e_2 *Werte sind, dann gilt:* $e_1 = e_2$*. Dies gilt unter folgender Voraussetzung: Alle Regeln* $l = r$ *und* $l' = r'$ *sind*

nicht überlappend: $\sigma(l) \neq \sigma(l')$ $\forall$ *Substitutionen* σ*, oder*

trivial überlappend: *falls* σ *existiert mit* $\sigma(l) = \sigma(l')$*, dann ist auch* $\sigma(r) = \sigma(r')$*.*

Spielt damit die Strategie keine Rolle? Doch, denn die LO-Strategie kann Werte berechnen für Ausdrücke, bei denen LI nicht terminiert.

Beispiel 5.11 (Berechnungsstärke)**.** Gegeben sei das folgende Programm.

```
f x = f (x + 1)
p x = True
```

Dann berechnen die Strategien für `p (f 0)` folgende Ableitungen:

LI: `p (f 0)` $\to$ `p (f (0 + 1))` $\to$ `p (f 1)` $\to$ `p (f (1 + 1))` $\to$ `p (f 2)` $\to$ ...

LO: `p (f 0)` $\to$ True

Ist LO damit grundsätzlich besser als LI? Leider nicht, denn:

1. LO ist aufwändiger zu implementieren, da die aktuellen Parameter unausgewerte Ausdrücke sein können (vgl. Diskussion zu call-by-name in Kapitel 3.4.2). Beachte: LI entspricht call-by-value, LO entspricht call-by-name.

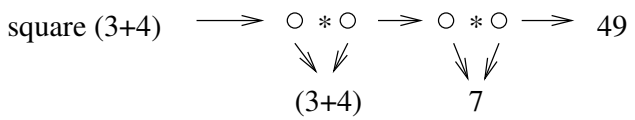
2. LO kann Teilausdrücke mehrfach auswerten, z. B. „`3 + 4`“ in obiger Ableitung.

Den letzten Nachteil kann man verbessern durch die sogenannte „faule“/verzögerte Auswertung (lazy evaluation, call by need). Die **faule Auswertung** basiert auf der folgenden Idee:

1. Berechne einen Teilausdruck erst dann, wenn er benötigt wird, z. B. Argumente von primitive Funktionen, `if-then-else`, `case`, ...

2. Berechne den Teilausdruck nicht komplett, sondern nur bis zu seiner **Kopfnormalform**: Dies ist ein Ausdruck, bei dem an der Wurzel keine definierte Funktion, sondern ein Konstruktor steht, z. B. „`0 : (conc [1] [2])`“, wobei „`:`“ hier die Wurzel ist.

3. Berechne jeden Ausdruck **höchstens einmal**. Die Implementierung hiervon geschieht durch **Graphdarstellung** der Ausdrücke (→ Graphreduktionsmaschinen).



Die Vorteile der faulen Auswertung sind:

- Überflüssige Auswertungen werden vermieden, dies führt zu einer optimalen Reduktion (Huet and Levy, 1979).

- Sie erlaubt das Rechnen mit unendlichen Datenstrukturen (→ Modularisierung: Trennung von Daten und Kontrolle).

Beispiel 5.12 (Lazy evaluation)**.** Wir definieren die Liste aller natürlichen Zahlen `(from 0)`, wobei

```
from n = n : from (n + 1)
```

Die ersten n Elemente einer Liste l (Kontrolle) erhält man mit `(take` n l`)`, wobei

```
take 0 xs             = []
take n (x:xs) | n > 0 = x : take (n - 1) xs
```

Nun ist folgende modulare Kombination möglich: `take 2 (from 0)` $\rightsquigarrow$ `[0,1]`

Schauen wir uns beim letzten Beispiel die faule Auswertung etwas genauer an:

```
take 2 (from 0)
→ take 2 (0 : from (0 + 1))
→ 0 : take (2 - 1) (from (0 + 1))
→ 0 : take 1 ((0 + 1): from ((0 + 1) + 1)
→ 0 : (0 + 1) : take (1 - 1) (from ((0 + 1) + 1))
→ 0 : (0 + 1) : []
→ 0 : 1 : []
```

Es ergibt sich der folgende Effekt: Obwohl der „Erzeuger“ `from` und der „Verbraucher“ `take` unabhängig implementiert sind (Modularität), laufen diese miteinander verschränkt ab.

Beispiel 5.13 (Liste aller Fibonacci-Zahlen)**.** Diese sind defininiert als

$$n_0 = 0 \qquad n_1 = 1 \qquad n_k = n_{k-1} + n_{k-2} \text{ für } k > 1$$

Wir implementieren die Idee, dass der Listenerzeuger zwei Parameter hat, nämlich die beiden vorherigen Fibonacci-Zahlen:

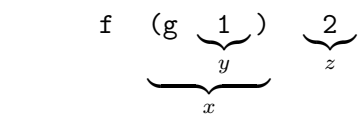
```
fibgen n1 n2 = n1 : fibgen n2 (n1 + n2)

fibs = fibgen 0 1  -- 0:1:1:2:3:5:8:13:...
```

Nun können die ersten 6 Fibonacci-Zahlen ermittelt werden als

```
take 6 fibs  ⇝ [0,1,1,2,3,5]
```

5.3.2 Formalisierung der faulen Auswertung

Die hier vorgestellte Formalisierung entspricht der Formalisierung von John Launchbury, die auf der POPL ’93 präsentiert wurde (Launchbury, 1993). Der bisherige Formalismus der Termersetzung ist für die Darstellung der faulen Auswertung unzureichend, da die „höchstens einmalige“ Auswertung eine graphähnliche Darstellung verlangt. Aus diesem Grund stellen wir nun Terme durch Zuordnung einer Variablen für jeden Teilausdruck dar, z. B. wird



umgeformt in `y = 1, z = 2, x = g y, f x z`

Diese Darstellung ist ausdrückbar mittels `let`-Ausdrücken, was ein Standardkonstrukt in funktionalen Sprachen ist:

`let` *decls* `in` *exp*

Beispiel 5.14 (`let`-Ausdrücke)**.**

```
f x y = let a = 2 + x
            b = x * y
        in  a * y + a * b
```

Hier wurde die **Layout-Regel** zur einfacheren Darstellung lokaler Deklarationen verwendet. Die Layout-Regel basiert auf folgenden Konventionen:

- Alle lokalen Deklarationen nach dem `let` beginnen in der gleichen Spalte.
- Falls eine neue Zeile rechts davon beginnt, handelt es sich um Fortsetzung der vorherigen Zeile.

Der Vorteil der Layout-Regel ist, dass Trennsymbole wie „`;`“ überflüssig sind und durch ein lesbares Code-Design ersetzt werden.

Mittels `let` können alle Ausdrücke in eine „flache Form“ transformiert werden.

Definition 5.8 (Flache Form)**.** *Die **flache Form** e^* eines Ausdrucks e ist definiert als:*

$$
\begin{array}{rll}
x^* = x & & \forall \textit{ Variablen } x \\
c^* = c & & \forall \textit{ Konstanten } c \\
(f\ e_1 \ldots e_n)^* = & \texttt{let} & x_1 = e_1^* \\
& & \vdots \\
& & x_n = e_n^* \\
& \texttt{in} & f\ x_1\ \ldots\ x_n
\end{array}
$$

wobei $x_1, \ldots, x_n$ neue Variablennamen und f eine definierte Funktion oder ein Konstruktor sind.

Die Erweiterung der Definition der flachen Form auf primitive Funktionen sowie `let`- bzw. `case`-Ausdrücke geschieht nach demselben Schema ($\rightsquigarrow$ Übung). Zusätzlich sind noch folgende Optimierungen der Übersetzung denkbar:

1. In $(f\ e_1 \ldots e_n)^*$ muss man nur neue Variablen für nicht-variable Argumente einführen. Dies könnte man auch auf Konstanten ausweiten, wobei unter Konstanten hier Zahlliterale wie `42` zu verstehen sind.
2. Transformiere geschachtelte `let`s:

$$
\begin{array}{lcll}
\texttt{let}\ x\ \texttt{=}\ (\texttt{let}\ y\ \texttt{=}\ e\ \texttt{in}\ e')\ \texttt{in}\ e'' & \Rightarrow & \texttt{let} & y = e \\
& & & x = e' \\
& & \texttt{in} & e'' \\
\texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ (\texttt{let}\ y\ \texttt{=}\ e'\ \texttt{in}\ e'') & \Rightarrow & \texttt{let} & x = e \\
& & & y = e' \\
& & \texttt{in} & e''
\end{array}
$$

Somit ergibt sich z.B. die folgende Transformation:

```
(f (g 1) y 2)* = let  x1 = 1
                      x2 = g x1
                      x3 = 2
                 in   f x2 y x3
```

Bei der flachen Form sind also alle Funktions- und Konstruktorargumente Variablen. Diese Variablen werden als Speicherzellen oder Graphknoten interpretiert.

$$\text{Speicher } M : \text{Variablen } \rightarrow \text{Ausdrücke}$$

Wir machen weiterhin die Annahme, dass jede Funktion f durch eine einzige Gleichung

$$f\ x_1 \ldots x_n = e \qquad \text{mit} \qquad e^* = e$$

definiert ist, d. h. alle Muster sind übersetzt in `if-then-else`- bzw. `case`-Ausdrücke und der Rumpf ist in flacher Form. Unter dieser Annahme ist das operationale Modell der faulen Auswertung leicht definierbar durch Inferenzregeln im Stil der natürlichen Semantik.

Basissprache

Die Basissprache ist hier

$$M : e \Downarrow M' : v$$

mit folgender Intuition:

> Im Speicherzustand M wird der Ausdruck e reduziert zu einem Wert v, wobei der Speicher zu M' modifiziert wird. Hierbei ist ein Wert ein spezieller Ausdruck (eine Konstante oder eine Konstruktoranwendung).

Wichtig ist hierbei, dass alle Ausdrücke in flacher Form sind. Unter diesen Annahmen können wir die faule Auswertung durch das folgende Regelsystem definieren.

Konstante/Konstruktor

$$M : C\ x_1 \ldots x_n \Downarrow M : C\ x_1 \ldots x_n$$

wobei C ein Konstruktor mit $n \geq 0$ Argumenten ist, d.h. der Ausdruck ist in Kopfnormalform.

Primitive Funktionen ($\oplus \in \{+, -, *, /, \ldots\}$)

$$\frac{M : e_1 \Downarrow M' : v_1 \quad M' : e_2 \Downarrow M'' : v_2}{M : e_1 \oplus e_2 \Downarrow M'' : v_1 \oplus v_2}$$

Hierbei ist wie bei der Interpretation von Ausdrücken (Kapitel 2.2.3) zu beachten, dass das linke $\oplus$ syntaktisch zu verstehen ist, während das rechte $\oplus$ die zugehörige semantische Funktion auf den Werten ist.

Funktionsanwendung

$$\frac{M : \sigma(e) \Downarrow M' : e'}{M : f\ y_1 \ldots y_n \Downarrow M' : e'}$$

falls $f\ x_1 \ldots x_n = e$ Regel für f ist und $\sigma = \{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}$.

`case`-Ausdruck

$$\frac{M : e \Downarrow M' : C\ y_1 \ldots y_k \quad M' : \sigma(e') \Downarrow M'' : e''}{M : \texttt{case}\ e\ \texttt{of}\ \ \ldots C\ x_1 \ldots x_k \to e' \ldots \Downarrow M'' : e''}$$

mit $\sigma = \{x_1 \mapsto y_1, \ldots, x_k \mapsto y_k\}$.

Variable

$$\frac{M : e \Downarrow M' : e'}{M : x \Downarrow M'[x/e'] : e'}$$

falls $M(x) = e$.

`let`-Ausdruck

$$\frac{M[y_1/\sigma(e_1)] \ldots [y_n/\sigma(e_n)] : \sigma(e) \Downarrow M' : e'}{M : \texttt{let}\ \ x_1 = e_1 \ldots x_n = e_n\ \ \texttt{in}\ \ e \Downarrow M' : e'}$$

mit $\sigma = \{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}$, wobei $y_1, \ldots, y_n$ neue („frische“) Variablen sind.

Anmerkungen

- Die erste Regel spezifiziert, dass Ausdrücke nur bis zur Kopfnormalform ausgewertet werden.
- Bei `case`-Ausdrücken und der Funktionsanwendung werden lediglich die Variablen umgesetzt ($\approx$ Referenzen).
- Bei der Variablen-Regel wird wirklich der Speicher modifiziert (ersetze Ausdruck durch sein Ergebnis), um so die faule Auswertung zu erreichen.
- Die `let`-Regel entspricht dem Speichern der lokalen Ausdrücke. Die Umbenennung der Variablen ist hier wichtig, da die Bezeichner sonst eventuell andere Bezeichner überdecken könnten.

5.3.3 Erkennung unendlicher Schleifen

Die Regel für Variablen kann verbessert werden, um manchmal unendliche Schleifen zu erkennen.

Beispiel 5.15 (Unendliche Schleife)**.** Die Auswertung von „`let x = 1 + x in x`“

führt zu einer Endlosableitung, beziehungsweise es existiert keine endliche Ableitung bezüglich unseres Inferenzsystems.

Um dieses Problem zu beheben, modifizieren wir die Variablenregel und führen eine neue Regel ein.

Variable'

$$\frac{M : e \Downarrow M' : e'}{M[x/e] : x \Downarrow M'[x/e'] : e'}$$

Loop

$$M : x \Downarrow \langle \text{Fehlschlag: Schleife} \rangle$$

falls $M(x)$ undefiniert ist.

Dann liefert die Auswertung des obigen Ausdrucks einen Fehlschlag, was natürlich nicht als Wert anzusehen ist, sondern einer Fehlermeldung entspricht. Die Erkennung dieses Fehlschlags wird in Haskell auch als **black hole detection** bezeichnet.