Kapitel 2

Ungetypte arithmetische Ausdrücke

In diesem Kapitel führen wir eine simple “Programmiersprache” für Ganzzahlen und Zeichenketten ein. Die Sprache ist so rudimentär und hat den Namen Programmiersprache eigentlich nicht verdient. Sie hilft uns aber, wichtige Konzepte der formalen Behandlung von Programmiersprachen zu verdeutlichen.

In unserer Metasprache seinen die ganzen Zahlen 0, 1, -1, 2, -2, ... und Zeichenketten `"hallo"`, `"bla"`, ... vorhanden. Mit *Metasprache* meinen wir die Sprache, in der wir unsere *Objektsprachen*, d.h. die uns interessierenden Fragmente von objektorientierten Programmiersprachen, definieren und analysieren. Diese Metasprache ist die Sprache der Mathematik, und in einigen Fällen die Programmiersprache, die wir für die Implementation eines Interpreters oder Typprüfers verwenden. In diesem Kapitel benutzen wir *Haskell* [?].

In der Mathematik wird die Menge der ganzen Zahlen mit $\mathbb{Z}$ bezeichnet, die Menge der Zeichenkette bezeichnen wir mit $\mathbb{S}$. In Haskell heißt der Typ der ganzen Zahlen `Int` und der Typ der Zeichenketten `String`. Für ganze Zahlen verwenden wir die Buchstaben i, j, k und für Zeichenketten s.

2.1 Externe Syntax

Die Sprache, die wir in diesem Kapitel vorstellen, behandelt einen kleinen Teil der gültigen Ausdrücke von JAVA. Dieser Teil kann durch folgende (BNF-ähnliche) Grammatik beschrieben werden.

$$
\begin{array}{lcll}
e & ::= & n & \text{konstante ganze Zahl 0, 1, -1, \ldots} \\
 & | & s & \text{konstante Zeichenkette, z.B. } \texttt{"hallo"} \\
 & | & e_1 + e_2 & \text{Addition} \\
 & | & e_1 - e_2 & \text{Subtraktion} \\
 & | & e.\texttt{length}() & \text{Länge der Zeichenkette } e \\
 & | & \texttt{Integer.toString}(e) & \text{Verwandlung der Zahl } e \text{ in eine Zeichenkette}
\end{array}
$$

Beispiel 2.1 *Folgende JAVA-Ausdrücke werden von der Grammatik erfasst.*

```
5 + 4
"hallo".length() + 1
Integer.toString(753).length()
"hallo" + 1.length()
```

Dabei ist der letzte Ausdruck "unsinnig" und wird vom JAVA-Compiler zurückgewiesen. Wie man die sinnvollen von den unsinnigen Ausdrücken unterscheidet, werden wir in Kapitel **??** *besprechen.*

2.2 Interne Syntax

Um knapp und präzise über Programmausdrücke reden zu können, führt man eine *interne* oder *abstrakte Syntax* ein. Manchmal dient sie auch dazu, eine gedachte Programmiersprache zu definieren und zu analysieren, ohne sich über eine genaue Definition der Syntax den Kopf zerbrechen zu müssen.

$$
\begin{array}{rcll}
t & ::= & n & \text{konstante ganze Zahl} \\
 & | & s & \text{konstante Zeichenkette} \\
 & | & t_1 \;\texttt{plus}\; t_2 & \text{Addition} \\
 & | & t_1 \;\texttt{minus}\; t_2 & \text{Subtraktion} \\
 & | & \texttt{len}\, t & \text{String-Länge} \\
 & | & \texttt{str}\, t & \text{Verwandlung in Zeichenkette}
\end{array}
$$

Diese rekursive Definition der gültigen Terme t lässt sich in Haskell durch den folgenden *rekursiven Datentyp* beschreiben.

```
data Term = CInt Int
          | CStr String
          | Plus Term Term
          | Minus Term Term
          | Len Term
          | Str Term
```

Das Symbol `Term` bezeichnet den neuen Typ der Terme, den wir durch die sechs Klauseln definieren. Die Symbole `CInt`, `CStr`, `Plus`, `Minus`, `Len` und `Str` sind sogenannte Konstruktoren, durch die Ausdrücke vom Typ `Term` erzeugt werden können. In unserem Fall haben sie je ein bis zwei Argumente vom Typ `Int`, `String` oder `Term`. Die dritte Klausel z.B. ist so zuverstehen: sind t_1 und t_2 Ausdrücke vom Typ `Term`, so ist `Plus` t_1 t_2 wieder ein Ausdruck vom Typ `Term`.

Beispiel 2.2 *Die JAVA-Ausdrücke aus dem letzten Bespiel lassen sich wie folgt in interne Syntax übersetzen und in Ausdrücke vom Typ* `Term` *übersetzen:*

```
5 plus 4                    Plus (CInt 5) (CInt 4)
(len "hallo") plus 1        Plus (Len (CStr "hallo")) (CInt 1)
len (str 753)               Len (Str (CInt 753))
"hallo" plus (len 1)        Plus (CStr ("hallo")) (Len 1)
```

2.2.1 Induktiv definierte Mengen

Die oben gegebene Grammatik der internen Syntax ist eine Kurzschreibweise für die folgende *induktive Definition* der Menge von Termen Tm.

Definition 2.1 (Terme, induktiv)

1. $\mathbb{Z} \subseteq \mathsf{Tm}$
2. $\mathbb{S} \subseteq \mathsf{Tm}$
3. *Sind* $t_1, t_2 \in \mathsf{Tm}$, *so auch* $t_1\ \texttt{plus}\ t_2,\ t_1\ \texttt{minus}\ t_2 \in \mathsf{Tm}$.
4. *Ist* $t \in \mathsf{Tm}$, *so sind* $\texttt{len}\, t,\ \texttt{str}\, t \in \mathsf{Tm}$.

Diese Definition kann auch durch folgende *Inferenzregeln* ausgedrückt werden:

Definition 2.2 (Terme, durch Inferenzregeln)

$$\frac{i \in \mathbb{Z}}{i \in \mathsf{Tm}} \qquad \frac{s \in \mathbb{S}}{s \in \mathsf{Tm}}$$

$$\frac{t_1 \in \mathsf{Tm} \qquad t_2 \in \mathsf{Tm}}{t_1\ \texttt{plus}\ t_2 \in \mathsf{Tm}} \qquad \frac{t_1 \in \mathsf{Tm} \qquad t_2 \in \mathsf{Tm}}{t_1\ \texttt{minus}\ t_2 \in \mathsf{Tm}}$$

$$\frac{t \in \mathsf{Tm}}{\texttt{len}\, t \in \mathsf{Tm}} \qquad \frac{t \in \mathsf{Tm}}{\texttt{str}\, t \in \mathsf{Tm}}$$

Mit Hilfe dieser Inferenzregeln lassen sich Beweisbäume oder *Herleitungen*, generieren, die belegen, warum ein bestimmter Ausdruck t in Tm ist (bzw. wie er da "hereingekommen" ist). Dabei ist jede Regel so zu lesen: Wenn die *Prämissen* über dem Balken erfüllt sind, so gilt die *Konklusion* unter dem Balken.

Beispiel 2.3 *Wir beweisen, das* `(len "hallo") plus 1` *ein Term unserer Sprache ist.*

$$\dfrac{\dfrac{\dfrac{\texttt{"hallo"} \in \mathbb{S}}{\texttt{"hallo"} \in \mathsf{Tm}}}{\texttt{len}\,\texttt{"hallo"} \in \mathsf{Tm}} \qquad \dfrac{1 \in \mathbb{Z}}{1 \in \mathsf{Tm}}}{(\texttt{len}\,\texttt{"hallo"})\ \texttt{plus}\ 1 \in \mathsf{Tm}}$$

Genauso kann man beweisen, dass der unsinnige Ausdruck `"hallo" plus (len 1)` $\in \mathsf{Tm}$.

Übung 2.1 *Geben Sie eine Herleitung von* `len (str "bla") minus (5 plus 3)` $\in \mathsf{Tm}$ *an.*

Die Menge Tm ist *definiert* als die *kleineste Menge*, die unter den Inferenzregeln abgeschlossen ist.

Wir können die Menge der Terme schrittweise von unten *konstruieren*, mit Hilfe der folgenden Definition.

Definition 2.3 (Terme, schrittweise) *Für jede natürliche Zahl n, definiere eine Menge T_n wie folgt:*

$$\begin{array}{lcl} T_0 & = & \emptyset \\ T_{n+1} & = & \mathbb{Z} \cup \mathbb{S} \\ & \cup & \{t_1 \texttt{ plus } t_2, t_1 \texttt{ minus } t_2 \mid t_1, t_2 \in T_n\} \\ & \cup & \{\texttt{len}\, t, \texttt{str}\, t \mid t \in T_n\} \end{array}$$

Schliesslich sei $T = \bigcup_{n\in\mathbb{N}} T_n$ der Limes der Folge $(T_n)_{n\in\mathbb{N}}$.

Übung 2.2 *Geben Sie einen Ausdruck t an mit $t \in T_3$ aber $t \notin T_2$.*

Satz 2.1 *Die Mengen T_n sind* kumulativ*, d.h. $T_n \subseteq T_{n+1}$ für beliebiges $n \in \mathbb{N}$.*

Übung 2.3 *Beweisen Sie Satz 2.1.*

Die Konstruktion der Menge der Terme ist korrekt, d.h. T ist die kleinste Menge, die unter den Inferenzregeln in Definition 2.2 abgeschlossen ist.

Satz 2.2 $T = \mathsf{Tm}$. *Das entspricht:*

1. *T ist unter den Regeln in Def. 2.2 abgeschlossen.*

2. *Sei T' unter den Inferenzregeln abgeschlossen. Dann $T \subseteq T'$.*

Beweis.

1. Beispielhaft zeigen wir das $t_1 \texttt{ plus } t_2 \in T$, falls $t_1, t_2 \in T$. Nach Voraussetzung gibt es Indices $n_1, n_2 \in \mathbb{N}$ mit $t_1 \in T_{n_1}$ und $t_2 \in T_{n_2}$. Sei $n = \max(n_1, n_2)$. Da die Folge $(T_m)_{m\in\mathbb{N}}$ kumulativ ist (Satz 2.1), sind $t_1, t_2 \in T_n$ und damit ist $t_1 \texttt{ plus } t_2 \in T_{n+1} \subseteq T$.

2. Es genügt zu zeigen, dass für alle Indices $n \in \mathbb{N}$ und alle Terme $t \in T_i$ auch $t \in T'$ gilt. Wir beweisen diese Aussage durch vollständige Induktion über n.

   Sei z.B. $\texttt{len}\, t \in T^n$. Dann ist $n > 1$ und $t \in T_{n-1}$. Nach Induktionsvoraussetzung ist $t \in T'$. Da T' unter den Inferenzregeln abgeschlossen ist, gilt auch $\texttt{len}\, t \in T'$.

□

Übung 2.4 *Vervollständigen Sie diesen Beweis für die noch nicht behandelten Fälle.*

2.2.2 Rekursion über induktive Mengen

In unserer Metasprache können wir nun Funktionen definieren, die Terme manipulieren. Beispielweise eine Funktion $|\cdot| : \mathsf{Tm} \to \mathbb{N}$, die die Größe eines Terms bestimmt. Wir definieren diese Funktion rekursiv, und durch Fallunterscheidung über die verschiedenen Termformen. Wir geben eine Definition in mathematischer Notation und eine in Haskell.

$$
\begin{array}{lll}
|\cdot| & : & \mathsf{Tm} \to \mathbb{N} \\
\\
|i| & = & 1 \\
|s| & = & 1 \\
|t_1 \texttt{ plus } t_2| & = & 1 + |t_1| + |t_2| \\
|t_1 \texttt{ minus } t_2| & = & 1 + |t_1| + |t_2| \\
|\texttt{len}\, t| & = & 1 + |t| \\
|\texttt{str}\, t| & = & 1 + |t|
\end{array}
$$

```
size :: Term -> Int

size (CInt i) = 1
size (CStr s) = 1
size (Plus  t1 t2) = 1 + size t1 + size t2
size (Minus t1 t2) = 1 + size t1 + size t2
size (Len t) = 1 + size t
size (Str t) = 1 + size t
```

Diese Definition ist dadurch gerechtfertigt, dass eine rekursive Verwendung immer nur mit "kleinerem" Argument auftritt. In der Definition der Größe von $t_1 \texttt{ plus } t_2$ z.B. wird sich nur auf die Größe der *Subterme* t_1 und t_2 bezogen. Diese Intuition können wir auch mathematisch präzise fassen. Wir definieren eine Folge von Funktionen $(\mathsf{size}_n)_{n\in\mathbb{N}}$:

$$
\begin{array}{lll}
\mathsf{size}_0 & : & T_0 \to \mathbb{N} \\
\ldots \\
\mathsf{size}_{n+1} & : & T_{n+1} \to \mathbb{N} \qquad \text{verwendet } \mathsf{size}_n : T_n \to \mathbb{N} \\
\ldots
\end{array}
$$

Jede dieser Funktion ist *nicht* rekursiv und *approximiert* die zu konstruierende Funktion $|\cdot| : T \to \mathbb{N}$, welche wir als Grenzwert der Folge erhalten.

Funktionen über Terme wie $|\cdot|$, die sich selbst nur mit Subtermen als Argument aufrufen, nennt man auch *strukturell rekursiv*. Eine weitere strukturell rekursive Funktion berechnet die Höhe eines Terms.

$$
\begin{array}{lll}
\mathsf{height} & : & \mathsf{Tm} \to \mathbb{N} \\
\\
\mathsf{height}(i) & = & 1 \\
\mathsf{height}(s) & = & 1 \\
\mathsf{height}(t_1 \texttt{ plus } t_2) & = & 1 + \max(\mathsf{height}(t_1), \mathsf{height}(t_2)) \\
\mathsf{height}(t_1 \texttt{ minus } t_2) & = & 1 + \max(\mathsf{height}(t_1), \mathsf{height}(t_2)) \\
\mathsf{height}(\texttt{len}\, t) & = & 1 + \mathsf{height}(t) \\
\mathsf{height}(\texttt{str}\, t) & = & 1 + \mathsf{height}(t)
\end{array}
$$

Übung 2.5 *Schreiben sie eine strukturell rekursive Funktion, die einen Ausdruck der internen Syntax in externe Syntax übersetzt.*

2.2.3 Beweise durch Induktion

Es gibt drei Arten, beweise für Aussagen über Terme t zu führen:

1. Induktion über die Termhöhe $\mathsf{height}(t) \in \mathbb{N}$.

2. Induktion über die Termgrösse $|t| \in \mathbb{N}$.

3. Strukturelle Induktion über die Konstruktion von $t \in \mathsf{Tm}$.

Alternative 3 vermeidet den Umweg über die natürlichen Zahlen und führt meistens zu einer klaren Beweisstruktur. Wann immer möglich, werden wir strukturelle Induktion verwenden.

Satz 2.3 *Für alle Terme* $t \in \mathsf{Tm}$ *gilt* $\mathsf{height}(t) \leq |t|$.

Beweis. Durch strukturelle Induktion über $t \in \mathsf{Tm}$. Wir müssen nun alle sechs Fälle abhandeln:

Fall $t = i$. Es gilt $\mathsf{height}(i) = 1 \leq 1 = |i|$.

Fall $t = s$. Analog.

Fall $t = t_1$ `plus` t_2. Nach Induktionsvoraussetzung ist $\mathsf{height}(t_n) \leq |t_n|$ für $n \in \{1, 2\}$. Somit gilt

$$\mathsf{height}(t) = 1 + \max(\mathsf{height}(t_1), \mathsf{height}(t_2)) \leq 1 + |t_1| + |t_2| = |t|.$$

Fall $t = t_1$ `minus` t_2. Analog.

Fall $t =$ `len` t'. Nach Induktionsvoraussetzung ist $\mathsf{height}(t') \leq |t'|$. Damit auch

$$\mathsf{height}(t) = 1 + \mathsf{height}(t') \leq 1 + |t'| = |t|.$$

Fall $t = \texttt{str}(t')$. Analog. □

Übung 2.6 *Beweisen Sie* $|t| \leq 2^{\mathsf{height}(t)} - 1$ *für alle Terme* $t \in \mathsf{Tm}$.

2.3 Semantik

Bis jetzt haben wir die Terme als reine syntaktische Objekte behandelt. Obwohl die Namen der Termkonstruktoren eine gewisse Bedeutung suggerieren, haben wir doch die Semantik nicht definiert.

Es gibt verschiedene Methoden, die Bedeutung von Programmkonstrukten mathematisch präzise zu erklären. Man unterscheidet zwischen

1. *operationeller Semantik*, welche für jeden Term beschreibt, wie er von einer abstrakten Maschine ausgewertet wird,

2. *denotationeller Semantik*, die Terme in mathematische Objekte übersetzt, und

3. *axiomatischer Semantik*, die das Programmverhalten durch logische Axiome beschreibt.

Wir werden uns ausschließlich mit der ersten Alternative befassen, da sie gleichzeitig einen Grundbaustein zur Implementation eines Interpreters darstellt.

In der Praxis findet man häufig eine von zwei weiteren Alternativen.

- Natürlich-sprachliche Semantik. Programmkonstrukte werden informell erklärt.
- Gar keine Semantik. Es gibt keine genaue Sprachbeschreibung. Die Bedeutung eines Programms ergibt sich, in dem man es kompiliert und ausführt.

Diese Ansätze haben entscheidende Schwächen: Falls keine genaue Spezifikation existiert, werden verschiedene Übersetzer ein- und dasselbe Programm mit großer Wahrscheinlichkeit leicht unterschiedlich interpretieren. Anwendungen, die unter einem Compiler entwickelt wurden, funktionieren mit einem anderen Compiler plötzlich nicht mehr. Die Programmiersprache ist nicht portabel.

2.3.1 Abstrakte Maschine

Wir definieren die *Auswertung* eines Programmausdrucks durch eine *abstrakte Maschine*. Die Maschine ist abstrakt in dem Sinne, dass wir konkrete Details ignorieren, z.B. wie Speicher repräsentiert ist etc. Das Ergebnis der Berechnung durch die Maschine ist ein *Wert*. In unserem Fall gibt es nur zwei verschiedene Kategorien von Werten.

$$\begin{array}{rcll} v & ::= & i & \text{Ganzzahl-Konstante} \\ & | & s & \text{String-Konstante} \end{array}$$

Der Zustand der Maschine wird vollständig bestimmt durch den Ausdruck, der gerade ausgewertet werden soll. Durch einen Auswertungsschritt wird die Maschine in den nächsten Zustand überführt, den wir wiederum durch einen Ausdruck charakterisieren. Die Zustandsübergangsrelation können wir daher als binäre Relation zwischen Termen schreiben.

$$t \longrightarrow t' \qquad t \text{ wertet in einem Schritt zu } t' \text{ aus}$$

Welche Terme sollen wir nun in Relation zueinander setzen? Zuerst einmal definieren wir, wie sich unsere Operationen `plus`, `minus`, `len`, `str` auf Werten verhalten.

Definition 2.4 (Auswertungssemantik I: Berechnungsregeln)

$$\frac{}{i_1 \;\texttt{plus}\; i_2 \longrightarrow (i_1 + i_2)}\ \text{E-PlusVV} \qquad \frac{}{i_1 \;\texttt{minus}\; i_2 \longrightarrow (i_1 - i_2)}\ \text{E-MinusVV}$$

$$\frac{}{\texttt{len}\, s \longrightarrow i_s}\ \text{E-LenV} \qquad \textit{wobei } i_s \textit{ die Länge von } s \textit{ bezeichnet}$$

$$\frac{}{\texttt{str}\, i \longrightarrow s_i}\ \text{E-StrV} \qquad \textit{wobei } s_i \textit{ die Dezimalnotation von } i \textit{ bezeichnet}$$

Beispiel 2.4

$$\begin{array}{lll} 5+4 & \longrightarrow & 9 \\ \texttt{len}\,\texttt{"hallo"} & \longrightarrow & 5 \\ \texttt{str}\,753 & \longrightarrow & \texttt{"753"} \end{array}$$

Noch ungeklärt ist, wie z.B. der Ausdruck $(5\ \texttt{plus}\ 4)\ \texttt{minus}\ (4\ \texttt{plus}\ 3)$ ausgewertet wird. Es fehlen noch Regeln, die angeben, was z.B. mit Termen $t_1\ \texttt{minus}\ t_2$ passieren soll, wenn t_1 oder t_2 keine Werte sind. Im Prinzip haben wir im obigen Beispiel zwei Möglichkeiten:

$$(5\ \texttt{plus}\ 4)\ \texttt{minus}\ (4\ \texttt{plus}\ 3) \longrightarrow 9\ \texttt{minus}\ (4\ \texttt{plus}\ 3) \longrightarrow 9\ \texttt{minus}\ 7 \longrightarrow 2,$$

oder

$$(5\ \texttt{plus}\ 4)\ \texttt{minus}\ (4\ \texttt{plus}\ 3) \longrightarrow (5\ \texttt{plus}\ 4)\ \texttt{minus}\ 7 \longrightarrow 9\ \texttt{minus}\ 7 \longrightarrow 2.$$

Wir könnten nun beide zulassen, dann wäre unsere Auswertung *nicht-deterministisch*. Wir bevorzugen jedoch eine *deterministische Auswertungsstrategie* und entscheiden uns (willkürlich) für die erste der beiden Alternativen.

Definition 2.5 (Auswertungssemantik II: Kongruenzregeln)

$$\frac{t_1 \longrightarrow t_1'}{t_1\ \texttt{plus}\ t_2 \longrightarrow t_1'\ \texttt{plus}\ t_2}\ \text{E-PLUSTT} \qquad \frac{t \longrightarrow t'}{i\ \texttt{plus}\ t \longrightarrow i\ \texttt{plus}\ t'}\ \text{E-PLUSVT}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ \texttt{minus}\ t_2 \longrightarrow t_1'\ \texttt{minus}\ t_2}\ \text{E-MINUSTT} \qquad \frac{t \longrightarrow t'}{i\ \texttt{minus}\ t \longrightarrow i\ \texttt{minus}\ t'}\ \text{E-MINUSVT}$$

$$\frac{t \longrightarrow t'}{\texttt{len}\,t \longrightarrow \texttt{len}\,t'}\ \text{E-LENT} \qquad \frac{t \longrightarrow t'}{\texttt{str}\,t \longrightarrow \texttt{str}\,t'}\ \text{E-STRT}$$

Die Kongruenzregeln haben je eine Hypothese: Z.B. liest sich die Regel E-LENT folgendermaßen: Wenn die Maschine t in einem Schritt zu t' auswerten könnte, dann ist der Übergang von $\texttt{len}\,t$ nach $\texttt{len}\,t'$ auch möglich. Die Möglichkeit eines Auswertungsschrittes können wir durch eine Herleitung belegen, an deren Anfang eine Berechnungsregel steht, gefolgt von Kongruenzregeln.

Beispiel 2.5

$$\cfrac{\cfrac{\overline{\texttt{str}\,753 \longrightarrow \texttt{"753"}}\ \text{E-STRV}}{\texttt{len}(\texttt{str}\,753) \longrightarrow \texttt{len}(\texttt{"753"})}\ \text{E-LENT}}{\texttt{len}(\texttt{str}\,753)\ \texttt{plus}\ (5\ \texttt{plus}\ 4) \longrightarrow \texttt{len}(\texttt{"753"})\ \texttt{plus}\ (5\ \texttt{plus}\ 4)}\ \text{E-PLUSTT}$$

Übung 2.7 *Für die folgenden Terme t, entscheiden Sie ob ein Auswertungsschritt zu einem Term t' möglich ist. Wenn ja, geben Sie eine Herleitungen für $t \longrightarrow t'$ an.*

1. $t = 5$ `minus` (`len` `"hallo"` `plus` (2 `minus` 3)).
2. $t =$ (`len` 753) `plus` (`str` `"hallo"`).
3. $t =$ `str`((4 `plus` 1) `minus` (`len` `"hallo"`)).

Definition 2.6 (Reduktion) *Die durch die Berechnungs- und Kongruenzregeln definierte Relation $\longrightarrow$ heißt* Ein-Schritt-Reduktion *oder* Ein-Schritt-Auswertung *und ist eine* small-step semantics. *Die Regeln nennen wir auch* Reduktionsregeln.

Alle Terme t, die nicht mit irgeneinem anderen Term t' in der Relation $t \longrightarrow t'$ stehen, können auch nicht weiter ausgewertet werden. Diese nennen wir *Normalformen.* Darunter fallen z.B. alle Werte v, aber auch unsinnige Terme wie Æ `minus` 5.

Definition 2.7 (Normalform) *Gibt es kein t' mit $t \longrightarrow t'$ so ist t in Normalform.*

Satz 2.4 *Jeder Wert v ist in Normalform.*

Beweis. Durch Überprüfen aller Auswertungsregeln. □

Unsere Auswertungsrelation ist in der Tat deterministisch.

Satz 2.5 (Auswertung ist deterministisch) *Wenn $t \longrightarrow t'$ und $t \longrightarrow t''$, dann ist $t' = t''$.*

Beweis. Durch Induktion nach $t \longrightarrow t'$. □

Definition 2.8 (Mehr-Schritt-Reduktion) *Wir definieren die Relationen:*

$\longrightarrow^+$	*transitive Hülle von* $\longrightarrow$
$\longrightarrow^*$	*reflexiv-transitive Hülle von* $\longrightarrow$

Übung 2.8 *Definieren Sie $\longrightarrow^+$ und $\longrightarrow^*$ durch Inferenzregeln. Dabei dürfen Sie $\longrightarrow$ als gegeben voraussetzen.*

Satz 2.6 (Mehr-Schritt-Auswertung ist deterministisch) *Sei t ein Term und u, u' Normalformen. Wenn $t \longrightarrow^* u$ und $t \longrightarrow^* u'$, dann $u = u'$.*

2.3.2 Natürliche Semantik

Im letzten Teil haben wir eine operationelle Semantik in Form einer *small-step* Auswertungsrelation kennengelernt. Dabei wurden die einzelnen Schritte spezifiziert, die eine abstrakte Maschine durchführen kann, um einen Ausdruck in Normalform überzuführen. Alternativ kann man die Semantik auch *natürlich* bzw. in Form einer *big-step* Auswertungsrelation definieren. Diese Relation gibt direkt an, zu welchem Wert ein Term auswertet.

Definition 2.9 (Big-step Semantik) *Die Auswertungsrelation $t \Downarrow v$ ist durch folgende Regeln gegeben.*

$$\frac{}{v \Downarrow v}\ \text{B-Value}$$

$$\frac{t_1 \Downarrow i_1 \qquad t_2 \Downarrow i_2}{t_1\ \texttt{plus}\ t_2 \Downarrow (i_1 + i_2)}\ \text{B-Plus} \qquad \frac{t_1 \Downarrow i_1 \qquad t_2 \Downarrow i_2}{t_1\ \texttt{minus}\ t_2 \Downarrow (i_1 - i_2)}\ \text{B-Minus}$$

$$\frac{t \Downarrow s}{\texttt{len}\, t \Downarrow i_s}\ \text{B-Len} \qquad \frac{t \Downarrow i}{\texttt{str}\, t \Downarrow s_i}\ \text{B-Str}$$

Dabei sind i_s und s_i wie oben definiert.

Für unsere Sprache sind small- und big-step Semantik äquivalent.

Satz 2.7 *Sei t ein Term und v ein Wert.*

1. *Wenn $t \Downarrow v$, dann $t \longrightarrow^* v$.*
2. *Wenn $t \longrightarrow^* v$, dann $t \Downarrow v$.*

Übung 2.9 *Beweisen Sie Satz 2.7. Teil 1 geht direkt mittels struktureller Induktion; für Teil 2 müssen Sie erst die folgende Hilfsaussage beweisen:*

$$\textit{Wenn } t \longrightarrow t' \textit{ und } t' \Downarrow v, \textit{ dann } t \Downarrow v.$$