5 Erweiterungen von While

Für die bisher betrachtete Sprache While gab es wegen der Einfachheit der Sprache bei der Modellierung der Semantik wenig Entscheidungsalternativen. Die entwickelten Semantiken bestehen aus „natürlichen“ Regeln, an denen wenig zu rütteln ist. Eine neue Semantik für eine Programmiersprache zu finden, hat aber viel mit Entwurfs- und Modellierungsentscheidungen zu tun. Deswegen werden in diesem Teil einige Erweiterungen für While entwickelt, die auch einige Vor- und Nachteile von Big-Step- und Small-Step-Semantiken aufzeigen werden.

Definition 12 (Modulare Erweiterung). Eine Erweiterung heißt *modular*, wenn man lediglich neue Regeln zur Semantik hinzufügen kann, ohne die bisherigen Regeln anpassen zu müssen. Mehrere modulare Erweiterungen können normalerweise problemlos kombiniert werden.

5.1 Nichtdeterminismus While_{ND}

Sowohl Big-Step- als auch Small-Step-Semantik für While sind deterministisch (Thm. 2 und 4). Die erste (modulare) Erweiterung While_{ND} führt eine neue Anweisung c_1 `or` c_2 ein, die nichtdeterministisch entweder c_1 oder c_2 ausführt. While_{ND}-Programme bestehen also aus folgenden Anweisungen:

$$\text{Com}\quad c \;::=\; \texttt{skip} \mid x \texttt{ := } a \mid c_0\texttt{; } c_1 \mid \texttt{if (}b\texttt{) then } c_1 \texttt{ else } c_2 \mid \texttt{while (}b\texttt{) do } c \mid c_1 \texttt{ or } c_2$$

Beispiel 8. Das Programm `x := 5 or x := 7` kann der Variablen `x` entweder den Wert 5 oder den Wert 7 zuweisen.

5.1.1 Big-Step-Semantik

Die Ableitungsregeln für $\langle \_,\ \_\rangle \Downarrow \_$ werden für das neue Konstrukt c_1 `or` c_2 um die beiden folgenden erweitert:

$$\textsc{Or1}_{\text{BS}}\colon \frac{\langle c_1,\ \sigma\rangle \Downarrow \sigma'}{\langle c_1 \texttt{ or } c_2,\ \sigma\rangle \Downarrow \sigma'} \qquad \textsc{Or2}_{\text{BS}}\colon \frac{\langle c_2,\ \sigma\rangle \Downarrow \sigma'}{\langle c_1 \texttt{ or } c_2,\ \sigma\rangle \Downarrow \sigma'}$$

Übung: Welche Ableitungsbäume hat das Programm $P \equiv$ `(x := 5) or (while (true) do skip)` in der Big-Step-Semantik?

5.1.2 Small-Step-Semantik

Die Small-Step-Semantik $\langle \_,\ \_\rangle \rightarrow_1 \langle \_,\ \_\rangle$ muss ebenfalls um Regeln für c_1 `or` c_2 ergänzt werden:

$$\textsc{Or1}_{\text{SS}}\colon \langle c_1 \texttt{ or } c_2,\ \sigma\rangle \rightarrow_1 \langle c_1,\ \sigma\rangle \qquad \textsc{Or2}_{\text{SS}}\colon \langle c_1 \texttt{ or } c_2,\ \sigma\rangle \rightarrow_1 \langle c_2,\ \sigma\rangle$$

Beispiel 9. Das Programm $P \equiv$ `(x := 5) or (while (true) do skip)` hat zwei maximale Ablei-

tungsfolgen:

$$\langle P,\ \sigma\rangle \to_1 \langle \texttt{x := 5},\ \sigma\rangle \to_1 \langle \texttt{skip},\ \sigma[\texttt{x}\mapsto 5]\rangle$$
$$\langle P,\ \sigma\rangle \to_1 \langle \texttt{while (true) do skip},\ \sigma\rangle$$
$$\to_1 \langle \texttt{if (true) then (skip; while (true) do skip) else skip},\ \sigma\rangle$$
$$\to_1 \langle \texttt{skip; while (true) do skip},\ \sigma\rangle \to_1 \langle \texttt{while (true) do skip},\ \sigma\rangle \to_1 \ldots$$

Im Vergleich zur Small-Step-Semantik unterdrückt die Big-Step-Semantik bei nichtdeterministischen Verzweigungen die nichtterminierenden Ausführungen. Insofern sind für nichtdeterministische Sprachen Big-Step- und Small-Step-Semantik nicht äquivalent: (Potenzielle) Nichttermination ist in der Big-Step-Semantik nicht ausdrückbar.

Übung: Welche der Beweise über die Small-Step- bzw. Big-Step-Semantik für While lassen sich auf While_{ND} übertragen?

- Determinismus von Big-Step- und Small-Step-Semantik (Thm. 2 und 4)
- Fortschritt der Small-Step-Semantik (Lem. 3)
- Äquivalenz von Big-Step- und Small-Step-Semantik (Kor. 10)

5.2 Parallelität While_{PAR}

Als Nächstes erweitern wir While um die Anweisung c_1 `||` c_2, die die Anweisungen c_1 und c_2 parallel ausführt, d.h., sowohl c_1 als auch c_2 werden ausgeführt, die Ausführungen können dabei aber verzahnt (interleaved) ablaufen.

Beispiel 10. Am Ende der Ausführung des Programms `x := 1 || (x := 2; x := x + 2)` kann `x` drei verschiedene Werte haben: 4, 1 und 3. Die möglichen verzahnten Ausführungen sind:

```
x := 1          |        x := 2        |        x := 2
      x := 2    |        x := x + 2    |    x := 1
      x := x + 2|    x := 1            |        x := x + 2
```

Diese Verzahnung lässt sich in der Small-Step-Semantik durch folgende neue Regeln modellieren:

$$\textsc{Par1}: \frac{\langle c_1,\ \sigma\rangle \to_1 \langle c_1',\ \sigma'\rangle}{\langle c_1\ \texttt{||}\ c_2,\ \sigma\rangle \to_1 \langle c_1'\ \texttt{||}\ c_2,\ \sigma'\rangle} \qquad \textsc{Par2}: \frac{\langle c_2,\ \sigma\rangle \to_1 \langle c_2',\ \sigma'\rangle}{\langle c_1\ \texttt{||}\ c_2,\ \sigma\rangle \to_1 \langle c_1\ \texttt{||}\ c_2',\ \sigma'\rangle}$$

$$\textsc{ParSkip1}: \langle \texttt{skip}\ \texttt{||}\ c,\ \sigma\rangle \to_1 \langle c,\ \sigma\rangle \qquad \textsc{ParSkip2}: \langle c\ \texttt{||}\ \texttt{skip},\ \sigma\rangle \to_1 \langle c,\ \sigma\rangle$$

Bemerkung. Anstelle der Regeln ParSkip1 und ParSkip2 könnte man auch die kombinierte Regel

$$\textsc{ParSkip}: \langle \texttt{skip}\ \texttt{||}\ \texttt{skip},\ \sigma\rangle \to_1 \langle \texttt{skip},\ \sigma\rangle$$

verwenden. Beide Varianten definieren die gleiche Semantik (im Sinne der Existenz unendlicher Ableitungsfolgen bzw. erreichbarer Endzustände) für While_{PAR}-Programme, jedoch sind einige Beweise mit den Regeln ParSkip1 und ParSkip2 technisch einfacher (siehe Übung).

Versucht man, eine entsprechende Erweiterung für die Big-Step-Semantik zu finden, stellt man fest, dass dies nicht möglich ist. Da c_1 und c_2 von c_1 `||` c_2 mit den Regeln der Big-Step-Semantik nur immer vollständig ausgewertet werden können, kann eine verschränkte Ausführung nicht angegeben werden.

5.3 Blöcke und lokale Variablen While_B

Bisher waren alle Variablen eines Programms global. Guter Stil in modernen Programmiersprachen ist aber, dass Variablen nur in dem Bereich sichtbar und zugreifbar sein sollen, in dem sie auch benötigt werden. Zum Beispiel werden Schleifenzähler für `for`-Schleifen üblicherweise im Schleifenkopf deklariert und sind nur innerhalb der Schleife zugreifbar.

Ein *Block* begrenzt den Sichtbarkeitsbereich einer lokalen Variablen x. Die Auswirkungen einer Zuweisung an x sollen sich auf diesen Block beschränken. Die neue Erweiterung While_B von While um Blöcke mit Deklarationen von lokalen Variablen führt die neue Block-Anweisung `{ var` x `=` a`;` c `}` ein. Semantisch soll sich dieser Block wie c verhalten, nur dass zu Beginn die Variable x auf den Wert von a initialisiert wird, nach Abarbeitung des Blocks aber immer noch den ursprünglichen Wert hat.

5.3.1 Big-Step-Semantik

Die Semantik $\langle \_, \_ \rangle \Downarrow \_$ wird mit folgender Regel erweitert:

$$\textsc{Block}_{\text{BS}}\colon \frac{\langle c,\ \sigma[x \mapsto \mathcal{A}\,[\![a]\!]\,\sigma] \rangle \Downarrow \sigma'}{\langle \texttt{\{ var } x \texttt{ = } a\texttt{; } c \texttt{ \}},\ \sigma \rangle \Downarrow \sigma'[x \mapsto \sigma(x)]}$$

Beispiel 11. Ableitungsbaum zu $P \equiv$ `{ var x = 0; { var y = 1; x := 5; y := x + y }; y := x }` im Startzustand $\sigma_1 = [\mathtt{x} \mapsto 10, \mathtt{y} \mapsto 20]$:

$$\dfrac{\dfrac{\dfrac{A \qquad \dfrac{}{\langle \texttt{y := x},\ \sigma_6 \rangle \Downarrow \sigma_7}\ \textsc{Ass}_{\text{BS}}}{\langle \texttt{\{ var y = 1; x := 5; y := x + y \}; y := x},\ \sigma_2 \rangle \Downarrow \sigma_7}\ \textsc{Seq}_{\text{BS}}}{\langle P,\ \sigma_1 \rangle \Downarrow \sigma_8}\ \textsc{Block}_{\text{BS}}$$

$$A\colon \dfrac{\dfrac{}{\langle \texttt{x := 5},\ \sigma_3 \rangle \Downarrow \sigma_4}\ \textsc{Ass}_{\text{BS}} \qquad \dfrac{}{\langle \texttt{y := x + y},\ \sigma_4 \rangle \Downarrow \sigma_5}\ \textsc{Ass}_{\text{BS}}}{\langle \texttt{\{ var y = 1; x := 5; y := x + y \}},\ \sigma_2 \rangle \Downarrow \sigma_6}\ \textsc{Block}_{\text{BS}}$$

		x	y
σ_1	$= [\mathtt{x} \mapsto 10, \mathtt{y} \mapsto 20]$	10	20
σ_2	$= \sigma_1[\mathtt{x} \mapsto 0]$	0	20
σ_3	$= \sigma_2[\mathtt{y} \mapsto 1]$	0	1
σ_4	$= \sigma_3[\mathtt{x} \mapsto 5]$	5	1
σ_5	$= \sigma_4[\mathtt{y} \mapsto \sigma_4(\mathtt{x}) + \sigma_4(\mathtt{y})]$	5	6
σ_6	$= \sigma_5[\mathtt{y} \mapsto \sigma_2(\mathtt{y})]$	5	20
σ_7	$= \sigma_6[\mathtt{y} \mapsto \sigma_6(\mathtt{x})]$	5	5
σ_8	$= \sigma_7[\mathtt{x} \mapsto \sigma_1(\mathtt{x})]$	10	5

5.3.2 Small-Step-Semantik

Blöcke sind in der Big-Step-Semantik sehr einfach, da der zusätzliche Speicherplatz, den man für die lokale Variable oder den ursprünglichen Wert benötigt, in der Regel $\textsc{Block}_{\text{BS}}$ versteckt werden kann. Die Small-Step-Semantik beschreibt immer nur einzelne Schritte, muss also den alten oder neuen Wert an einer geeigneten Stelle speichern. Dafür gibt es im Wesentlichen zwei Möglichkeiten:

1. Man ersetzt den Zustand durch einen Stack, der die vergangenen Werte speichert. Alle bisherigen Anweisungen ändern nur die obersten Werte, Blöcke legen zu Beginn neue Werte auf den Stack und nehmen sie am Ende wieder herunter.
2. Man speichert einen Teil der Zustandsinformation in der Programmsyntax selbst.

Im Folgenden wird die zweite, modulare Variante vorgestellt. Die neuen Regeln sind:

$$\textsc{Block1}_{SS}\colon \frac{\langle c,\ \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma]\rangle \to_1 \langle c',\ \sigma'\rangle}{\langle \texttt{\{ var } x \texttt{ = } a\texttt{; } c \texttt{ \}},\ \sigma\rangle \to_1 \langle \texttt{\{ var } x \texttt{ = } \mathcal{N}^{-1}[\![\sigma'(x)]\!]\texttt{; } c' \texttt{ \}},\ \sigma'[x \mapsto \sigma(x)]\rangle}$$

$$\textsc{Block2}_{SS}\colon \langle \texttt{\{ var } x \texttt{ = } a\texttt{; skip \}},\ \sigma\rangle \to_1 \langle \texttt{skip},\ \sigma\rangle$$

Beispiel 12. Ableitungsfolge zu $P \equiv$ `{ var x = 0; { var y = 1; x := 5; y := x + y }; y := x }` im Startzustand $\sigma = [\mathtt{x} \mapsto 10, \mathtt{y} \mapsto 20]$:

$$\begin{aligned}
\langle P,\ \sigma\rangle &\to_1 \langle \texttt{\{ var x = 5; \{ var y = 1; y := x + y \}; y := x \}},\ \sigma\rangle \\
&\to_1 \langle \texttt{\{ var x = 5; \{ var y = 6; skip \}; y := x \}},\ \sigma\rangle \\
&\to_1 \langle \texttt{\{ var x = 5; skip; y := x \}},\ \sigma\rangle \to_1 \langle \texttt{\{ var x = 5; y := x \}},\ \sigma\rangle \\
&\to_1 \langle \texttt{\{ var x = 5; skip \}},\ \sigma[\mathtt{y} \mapsto 5]\rangle \to_1 \langle \texttt{skip},\ \sigma[\mathtt{y} \mapsto 5]\rangle
\end{aligned}$$

5.4 Prozeduren

In diesem Abschnitt erweitern wir die While_B-Sprache um Prozeduren bzw. Funktionen. Waren die bisherigen Semantiken für While mit Erweiterungen meist ohne große Designentscheidungen, so gibt es bei Prozeduren mehrere Modellierungsmöglichkeiten mit unterschiedlicher Semantik. Wir beginnen mit der einfachsten Form, bei der Prozeduren quasi textuell an die Aufrufstelle kopiert werden und ohne Parameter auskommen, ähnlich zu Makros.[2]

5.4.1 Prozeduren ohne Parameter While_{PROC}

Die Syntax von While_{PROC} muss dazu Deklarationsmöglichkeiten für und Aufrufe von Prozeduren bereitstellen. Ein Programm P besteht ab sofort aus einer Anweisung c und einer Liste von Prozedurdeklarationen der Form (p, c), wobei p den Namen der Prozedur und c den Rumpf der Prozedur beschreibt. Wir nehmen im Folgenden immer an, dass die Prozedurnamen in der Deklarationsliste eindeutig sind. Die neue Anweisung `call` p ruft die Prozedur p auf.

Beispiel 13. (`sum`, `if (i == 0) then skip else (x := x + i; i := i - 1; call sum)`) deklariert eine Prozedur `sum`. Damit berechnet `x := 0; call sum` die Summe der ersten $\sigma(\mathtt{i})$ Zahlen, falls $\sigma(\mathtt{i}) \geq 0$ ist.

Wenden wir uns nun als erstes der Big-Step-Semantik zu. Diese braucht für die Aufrufregel die Prozedurdeklarationen. Deswegen ändern wir den Typ der Big-Step-Auswertungsrelation so, dass die Deklarationen als eine Umgebung P durch alle Regeln durchgeschleift werden:

$$\_ \vdash \langle \_, \_ \rangle \Downarrow \_ \subseteq \mathsf{PDecl}^* \times (\mathsf{Com} \times \Sigma) \times \Sigma$$

Entsprechend müssen alle bisherigen Regeln angepasst werden:

$$\textsc{Skip}^{\mathrm{P}}_{\mathrm{BS}}\colon P \vdash \langle \mathtt{skip},\ \sigma \rangle \Downarrow \sigma \qquad \textsc{Ass}^{\mathrm{P}}_{\mathrm{BS}}\colon P \vdash \langle x\ \mathtt{:=}\ a,\ \sigma \rangle \Downarrow \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma]$$

$$\textsc{Seq}^{\mathrm{P}}_{\mathrm{BS}}\colon \frac{P \vdash \langle c_0,\ \sigma \rangle \Downarrow \sigma' \qquad P \vdash \langle c_1,\ \sigma' \rangle \Downarrow \sigma''}{P \vdash \langle c_0\ \mathtt{;}\ c_1,\ \sigma \rangle \Downarrow \sigma''}$$

$$\textsc{IfTT}^{\mathrm{P}}_{\mathrm{BS}}\colon \frac{\mathcal{B}[\![b]\!]\sigma = \mathbf{tt} \qquad P \vdash \langle c_0,\ \sigma \rangle \Downarrow \sigma'}{P \vdash \langle \mathtt{if\ (}b\mathtt{)\ then}\ c_0\ \mathtt{else}\ c_1,\ \sigma \rangle \Downarrow \sigma'}$$

$$\textsc{IfFF}^{\mathrm{P}}_{\mathrm{BS}}\colon \frac{\mathcal{B}[\![b]\!]\sigma = \mathbf{ff} \qquad P \vdash \langle c_1,\ \sigma \rangle \Downarrow \sigma'}{P \vdash \langle \mathtt{if\ (}b\mathtt{)\ then}\ c_0\ \mathtt{else}\ c_1,\ \sigma \rangle \Downarrow \sigma'}$$

$$\textsc{WhileFF}^{\mathrm{P}}_{\mathrm{BS}}\colon \frac{\mathcal{B}[\![b]\!]\sigma = \mathbf{ff}}{P \vdash \langle \mathtt{while\ (}b\mathtt{)\ do}\ c,\ \sigma \rangle \Downarrow \sigma}$$

$$\textsc{WhileTT}^{\mathrm{P}}_{\mathrm{BS}}\colon \frac{\mathcal{B}[\![b]\!]\sigma = \mathbf{tt} \qquad P \vdash \langle c,\ \sigma \rangle \Downarrow \sigma' \qquad P \vdash \langle \mathtt{while\ (}b\mathtt{)\ do}\ c,\ \sigma' \rangle \Downarrow \sigma''}{P \vdash \langle \mathtt{while\ (}b\mathtt{)\ do}\ c,\ \sigma \rangle \Downarrow \sigma''}$$

$$\textsc{Block}^{\mathrm{P}}_{\mathrm{BS}}\colon \frac{P \vdash \langle c,\ \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma] \rangle \Downarrow \sigma'}{P \vdash \langle \mathtt{\{\ var}\ x\ \mathtt{=}\ a\mathtt{;}\ c\ \mathtt{\},}\ \sigma \rangle \Downarrow \sigma'[x \mapsto \sigma(x)]}$$

[2] Makros werden üblicherweise durch einen Präprozessor *statisch* im Text ersetzt, der Compiler oder die Semantik sehen von den Makros selbst nichts. Unsere Prozeduren dagegen werden erst *zur Laufzeit* eingesetzt.

Außerdem brauchen wir noch eine neue Regel für den Prozeduraufruf:

$$\textsc{Call}^{\text{P}}_{\text{BS}}\colon \frac{(p, c) \in P \qquad P \vdash \langle c,\ \sigma\rangle \Downarrow \sigma'}{P \vdash \langle \texttt{call}\ p,\ \sigma\rangle \Downarrow \sigma'}$$

Die Small-Step-Semantik ist für diese Variante der Prozeduren genauso einfach. Wie bei der Big-Step-Semantik erweitern wir $\langle \_,\ \_\rangle \rightarrow_1 \langle \_,\ \_\rangle$ um die Prozedurdeklarationsumgebung P, die Regeln wiederholen wir hier nicht nochmals. Für die Aufrufanweisung `call` p ergibt sich folgende Regel:

$$\textsc{Call}^{\text{P}}_{\text{SS}}\colon \frac{(p, c) \in P}{P \vdash \langle \texttt{call}\ p,\ \sigma\rangle \rightarrow_1 \langle c,\ \sigma\rangle}$$

Beispiel 14. Sei P die Liste, die nur die Prozedur `sum` wie im letzten Beispiel deklariert. Der Ableitungsbaum für `call sum` im Anfangszustand $\sigma_1 = [\mathtt{i} \mapsto 2, \mathtt{x} \mapsto 0]$ ist:

$\sigma_1 = [\mathtt{i} \mapsto 2, \mathtt{x} \mapsto 0]$
$\sigma_2 = [\mathtt{i} \mapsto 1, \mathtt{x} \mapsto 2]$
$\sigma_3 = [\mathtt{i} \mapsto 0, \mathtt{x} \mapsto 3]$

$$\dfrac{(\texttt{sum}, c_{\texttt{sum}}) \in P \qquad \dfrac{\dots \qquad \dfrac{\dots \qquad \dfrac{\dots \qquad \dfrac{\dots \qquad \dfrac{}{P \vdash \langle \texttt{skip},\ \sigma_3\rangle \Downarrow \sigma_3}}{P \vdash \langle c_{\texttt{sum}},\ \sigma_3\rangle \Downarrow \sigma_3}}{P \vdash \langle \texttt{call sum},\ \sigma_3\rangle \Downarrow \sigma_3}}{P \vdash \langle c_{\texttt{else}},\ \sigma_2\rangle \Downarrow \sigma_3}}{P \vdash \langle c_{\texttt{sum}},\ \sigma_2\rangle \Downarrow \sigma_3}}{P \vdash \langle \texttt{call sum},\ \sigma_2\rangle \Downarrow \sigma_3}}{P \vdash \langle c_{\texttt{else}},\ \sigma_1\rangle \Downarrow \sigma_3}}{P \vdash \langle c_{\texttt{sum}},\ \sigma_1\rangle \Downarrow \sigma_3}}{P \vdash \langle \texttt{call sum},\ \sigma_1\rangle \Downarrow \sigma_3}$$

Übung: Welche Ableitungsfolge ergibt sich in der Small-Step-Semantik für dieses Beispiel?

Obwohl die Semantik-Regeln einfach sind, ist diese Modellierung von Prozeduren aus folgenden Gründen nicht zufriedenstellend:

1. Es gibt keine Parameter, Wertüber- und -rückgabe ist nur über globale Variablen möglich.
2. Die Variablen in der Prozedur sind nicht statisch an globale Variablen gebunden, sondern werden *dynamisch* gebunden. Wenn ein umgebender Block eine lokale Variable deklariert, deren Namen in der Prozedur verwendet wird, so arbeitet die Prozedur beim Aufruf innerhalb des Blocks mit der lokalen Variable, außerhalb aber mit der globalen. Damit werden Abstraktionen unmöglich.

5.4.2 Prozeduren mit einem Parameter While$_{PROCP}$

Wegen der obigen Nachteile wollen wir nun noch eine Modellierung mit expliziten Parametern und statisch gebundenen Variablen ausarbeiten. Diese Änderung ist nicht modular, weil wir dafür die Zustandsmodellierung ändern müssen, so dass Variablen je nach Bedarf an andere Speicherstellen gebunden werden können.

Definition 13 (Speicher, Variablenumgebung). Der *Speicher* (*store*) s ist eine Funktion von *Speicherstellen* (*locations*) auf Werte. Eine *Variablenumgebung* E ordnet jeder Variablen eine Speicherstelle (location) zu, hat also den Typ $\mathsf{Var} \Rightarrow \mathsf{Loc}$.

Ein Zugriff auf eine Variable x erfolgt nun dadurch, dass

1. Die der Variablen x zugeordnete Speicherstelle $E(x)$ ermittelt wird, und
2. Im Speicher s auf die Stelle $E(x)$ – mit dem gespeicherten Wert $s(E(x))$ – zugegriffen wird.

Der Einfachheit halber sei $\mathsf{Loc} = \mathbb{Z}$. Neben der Speicherung der Werte der Speicherstellen muss ein Speicher auch noch vermerken, welche Speicherstelle die nächste unbenutzte ist. Demnach ist ein Speicher vom Typ

$$\mathsf{Store} \equiv \mathsf{Loc} \cup \{\mathsf{next}\} \Rightarrow \mathbb{Z},$$

wobei unter next die nächste freie Speicherzelle vermerkt ist.[3]

Definition 14 (Programm). Ein Programm der neuen Sprache While_{PROCP} besteht aus

1. einer Liste P von Prozedurdeklarationen,
2. der auszuführenden Anweisung und
3. einer Liste V der globalen Variablen, die von den Prozeduren und der Anweisung verwendet werden.

Definition 15 (Initiale Variablenumgebung, initialer Zustand). Die initiale Variablenumgebung E_0 ordnet den globalen Variablen die ersten $|V|$ Speicherstellen, d.h. von 0 bis $|V| - 1$, zu. Der initiale Zustand muss unter next die nächste freie Speicherstelle $|V|$ speichern.

Da Prozeduren nun auch einen Parameter bekommen und einen Rückgabewert berechnen sollen, müssen auch Prozedurdeklarationen und Aufrufe angepasst werden. Konzeptuell kann unser Ansatz auch auf mehrere Parameter- oder Rückgabewerte erweitert werden. Wegen der zusätzlichen formalen Komplexität betrachten wir hier aber nur Prozeduren mit einem Parameter.

Definition 16 (Prozedurdeklaration). Eine Prozedurdeklaration besteht nun aus

1. dem Prozedurnamen p,
2. dem Parameternamen x und
3. dem Rumpf der Prozedur als Anweisung.

Den Rückgabewert muss jede Prozedur in die spezielle (prozedurlokale) Variable `result` schreiben. Damit hat jede Prozedur automatisch zwei lokale Variablen: Den Parameter und die Rückgabevariable `result`.

Ein Aufruf hat nun die Form y `<- call` $p(a)$, wobei p der Prozedurname, a der arithmetische Ausdruck, dessen Wert an den Parameter übergeben wird und y die Variable ist, die den Rückgabewert aufnimmt.

Beispiel 15. Gegeben sei die Deklaration der Prozedur `sum2` mit Parameter `i` und Rumpf
`if (i == 0) then result := 1 else (result <- call sum2(i - 1); result := result + i)`
Der Aufruf `x <- call sum2(10)` speichert in der Variablen x die Summe der ersten 10 Zahlen.

[3] Da wir $\mathsf{Loc} = \mathbb{Z}$ gewählt haben, genügt uns dieser einfache Typ, da $s(\mathsf{next}) \in \mathsf{Loc}$ gelten muss. Im allgemeinen Fall wäre $\mathsf{Store} \equiv (\mathsf{Loc} \Rightarrow \mathbb{Z}) \times \mathsf{Loc}$, was die Syntax aufwändiger machte.

Für die Big-Step-Semantik muss die Auswertungsrelation wieder erweitert werden: Wir brauchen zusätzlich die globale Umgebung E_0 und die aktuelle Umgebung E, die den Variablen Speicherstellen zuordnen. Zum Programmstart sind diese beiden gleich, im Laufe der Ausführung kann sich E aber ändern. Außerdem gibt es keinen Zustand σ mehr, sondern nur noch einen globalen Speicher s. Damit hat die Auswertungsrelation folgende Form:

$$P, E_0, E \vdash \langle c,\ s\rangle \Downarrow s'$$

Wie schon mit P geschehen, müssen die Umgebungen E_0 und E durch alle Regeln durchgeschleift werden. Variablenzugriffe müssen jetzt über E und s erfolgen.

Definition 17 (Big-Step-Semantik für Prozeduren mit einem Parameter).
Die geänderten Regeln für die Auswertungsrelation sehen wie folgt aus:

$$\textsc{Skip}^{\text{P1}}_{\text{BS}}\colon P, E_0, E \vdash \langle \texttt{skip},\ s\rangle \Downarrow s \qquad \textsc{Ass}^{\text{P1}}_{\text{BS}}\colon P, E_0, E \vdash \langle x\ \texttt{:=}\ a,\ s\rangle \Downarrow s[E(x) \mapsto \mathcal{A}[\![a]\!](s \circ E)]$$

$$\textsc{Seq}^{\text{P1}}_{\text{BS}}\colon \frac{P, E_0, E \vdash \langle c_0,\ s\rangle \Downarrow s' \qquad P, E_0, E \vdash \langle c_1,\ s'\rangle \Downarrow s''}{P, E_0, E \vdash \langle c_0\texttt{;}\ c_1,\ s\rangle \Downarrow s''}$$

$$\textsc{IfTT}^{\text{P1}}_{\text{BS}}\colon \frac{\mathcal{B}[\![b]\!](s \circ E) = \mathbf{tt} \qquad P, E_0, E \vdash \langle c_0,\ s\rangle \Downarrow s'}{P, E_0, E \vdash \langle \texttt{if (}b\texttt{) then } c_0 \texttt{ else } c_1,\ s\rangle \Downarrow s'}$$

$$\textsc{IfFF}^{\text{P1}}_{\text{BS}}\colon \frac{\mathcal{B}[\![b]\!](s \circ E) = \mathbf{ff} \qquad P, E_0, E \vdash \langle c_1,\ s\rangle \Downarrow s'}{P, E_0, E \vdash \langle \texttt{if (}b\texttt{) then } c_0 \texttt{ else } c_1,\ s\rangle \Downarrow s'}$$

$$\textsc{WhileFF}^{\text{P1}}_{\text{BS}}\colon \frac{\mathcal{B}[\![b]\!](s \circ E) = \mathbf{ff}}{P, E_0, E \vdash \langle \texttt{while (}b\texttt{) do } c,\ s\rangle \Downarrow s}$$

$$\textsc{WhileTT}^{\text{P1}}_{\text{BS}}\colon \frac{\mathcal{B}[\![b]\!](s \circ E) = \mathbf{tt} \qquad P, E_0, E \vdash \langle c,\ s\rangle \Downarrow s' \qquad P, E_0, E \vdash \langle \texttt{while (}b\texttt{) do } c,\ s'\rangle \Downarrow s''}{P, E_0, E \vdash \langle \texttt{while (}b\texttt{) do } c,\ s\rangle \Downarrow s''}$$

$$\textsc{Block}^{\text{P1}}_{\text{BS}}\colon \frac{P, E_0, E[x \mapsto s(\mathsf{next})] \vdash \langle c,\ s[s(\mathsf{next}) \mapsto \mathcal{A}[\![a]\!](s \circ E), \mathsf{next} \mapsto s(\mathsf{next}) + 1]\rangle \Downarrow s'}{P, E_0, E \vdash \langle \texttt{\{ var } x \texttt{ = } a\texttt{; } c \texttt{ \}},\ s\rangle \Downarrow s'[\mathsf{next} \mapsto s(\mathsf{next})]}$$

$$\textsc{Call}^{\text{P1}}_{\text{BS}}\colon \frac{(p, x, c) \in P \qquad \begin{array}{c} P, E_0, E_0[x \mapsto s(\mathsf{next}), \texttt{result} \mapsto s(\mathsf{next}) + 1] \vdash \\ \langle c,\ s[s(\mathsf{next}) \mapsto \mathcal{A}[\![a]\!](s \circ E), \mathsf{next} \mapsto s(\mathsf{next}) + 2]\rangle \Downarrow s' \end{array}}{P, E_0, E \vdash \langle y\ \texttt{<- call } p\texttt{(}a\texttt{)},\ s\rangle \Downarrow s'[E(y) \mapsto s'(s(\mathsf{next}) + 1), \mathsf{next} \mapsto s(\mathsf{next})]}$$

Die Regeln $\textsc{Block}^{\text{P1}}_{\text{BS}}$ und $\textsc{Call}^{\text{P1}}_{\text{BS}}$ allozieren nun explizit neuen Speicher für die lokale Variable bzw. den Paramter und `result`. Nach der Ausführung setzen sie den `next`-Zeiger auf den Wert vor Beginn der Ausführung zurück. Dies ist möglich, weil neue Variablen (und damit neuer Speicher) nur strukturiert durch Blöcke bzw. Prozeduraufrufe alloziert werden, d.h., der Speicher wird stack-artig verwendet. Anschaulich ergibt sich folgende Speicheraufteilung:

wobei

G globale Variablen
L lokale Variablen
P Parameter
R Rückgabewert `result`

Beispiel 16. Sei P die Prozedurliste, die nur die Prozedur `sum2` vom letzten Beispiel deklariert, und das Hauptprogramm $c \equiv$ `x <- call sum2(2)`. Die Liste V der globalen Variablen ist dann [x]. Damit

ergibt sich die initiale Variablenumgebung E_0 zu $[\mathtt{x} \mapsto 0]$ und der Anfangsspeicher $s_0 \equiv [0 \mapsto ?, \mathsf{next} \mapsto 1]$, wobei der Anfangswert von `x` einen beliebigen Wert ? hat. Der Ableitungsbaum für c ist:

$$\dfrac{(\mathtt{sum2}, \mathtt{i}, c_{\mathtt{sum2}}) \in P \qquad \dfrac{\mathcal{B}[\![\mathtt{i == 0}]\!](s_1 \circ E_1) = \mathbf{ff} \qquad \dfrac{A \qquad \overline{P, E_0, E_1 \vdash \langle \mathtt{result := result + i},\ s_7\rangle \Downarrow s_8}}{P, E_0, E_1 \vdash \langle c_{\mathtt{else}},\ s_1\rangle \Downarrow s_8}}{P, E_0, E_1 \vdash \langle c_{\mathtt{sum2}},\ s_1\rangle \Downarrow s_8}}{P, E_0, E_0 \vdash \langle c,\ s_0\rangle \Downarrow s_9}$$

$$\text{A:}\ \dfrac{(\mathtt{sum2}, \mathtt{i}, c_{\mathtt{sum2}}) \in P \qquad \dfrac{\mathcal{B}[\![\mathtt{i == 0}]\!](s_2 \circ E_2) = \mathbf{ff} \qquad \dfrac{B \qquad \overline{P, E_0, E_2 \vdash \langle \mathtt{result := result + i},\ s_5\rangle \Downarrow s_6}}{P, E_0, E_2 \vdash \langle c_{\mathtt{else}},\ s_2\rangle \Downarrow s_6}}{P, E_0, E_2 \vdash \langle c_{\mathtt{sum2}},\ s_2\rangle \Downarrow s_6}}{P, E_0, E_1 \vdash \langle \mathtt{result <- call\ sum2(i\ -\ 1)},\ s_1\rangle \Downarrow s_7}$$

$$\text{B:}\ \dfrac{(\mathtt{sum2}, \mathtt{i}, c_{\mathtt{sum2}}) \in P \qquad \dfrac{\mathcal{B}[\![\mathtt{i == 0}]\!](s_3 \circ E_3) = \mathbf{tt} \qquad \overline{P, E_0, E_3 \vdash \langle \mathtt{result := 0},\ s_3\rangle \Downarrow s_4}}{P, E_0, E_3 \vdash \langle c_{\mathtt{sum2}},\ s_3\rangle \Downarrow s_4}}{P, E_0, E_2 \vdash \langle \mathtt{result <- call\ sum2(i\ -\ 1)},\ s_2\rangle \Downarrow s_5}$$

wobei

Variablenumgebung:	**Belegung:**	`x`	`i`	`result`
$E_0 = [\mathtt{x} \mapsto 0]$		0		
$E_1 = E_0[\mathtt{i} \mapsto s_0(\mathsf{next}), \mathtt{result} \mapsto s_0(\mathsf{next}) + 1]$		0	1	2
$E_2 = E_0[\mathtt{i} \mapsto s_1(\mathsf{next}), \mathtt{result} \mapsto s_1(\mathsf{next}) + 1]$		0	3	4
$E_3 = E_0[\mathtt{i} \mapsto s_2(\mathsf{next}), \mathtt{result} \mapsto s_2(\mathsf{next}) + 1]$		0	5	6

Speicher:	**Werte:** next	0	1	2	3	4	5	6
$s_0 = [0 \mapsto ?, \mathsf{next} \mapsto 1]$	1	?						
$s_1 = s_0[1 \mapsto \mathcal{A}[\![2]\!](s_0 \circ E_0), \mathsf{next} \mapsto 3]$	3	?	2	?				
$s_2 = s_1[3 \mapsto \mathcal{A}[\![\mathtt{i\ -\ 1}]\!](s_1 \circ E_1), \mathsf{next} \mapsto 5]$	5	?	2	?	1	?		
$s_3 = s_2[5 \mapsto \mathcal{A}[\![\mathtt{i\ -\ 1}]\!](s_2 \circ E_2), \mathsf{next} \mapsto 7]$	7	?	2	?	1	?	0	?
$s_4 = s_3[E_3(\mathtt{result}) \mapsto \mathcal{A}[\![0]\!](s_3 \circ E_3)]$	7	?	2	?	1	?	0	0
$s_5 = s_4[E_2(\mathtt{result}) \mapsto s_5(s_2(\mathsf{next}) + 1), \mathsf{next} \mapsto 5]$	5	?	2	?	1	0	*0*	*0*
$s_6 = s_5[E_2(\mathtt{result}) \mapsto \mathcal{A}[\![\mathtt{result\ +\ i}]\!](s_5 \circ E_2)]$	5	?	2	?	1	1	*0*	*0*
$s_7 = s_6[E_1(\mathtt{result}) \mapsto s_6(s_1(\mathsf{next}) + 1), \mathsf{next} \mapsto 3]$	3	?	2	1	*1*	*1*	*0*	*0*
$s_8 = s_7[E_1(\mathtt{result}) \mapsto \mathcal{A}[\![\mathtt{result\ +\ i}]\!](s_7 \circ E_1)]$	3	?	2	3	*1*	*1*	*0*	*0*
$s_9 = s_8[E_0(\mathtt{x}) \mapsto s_8(s_0(\mathsf{next}) + 1), \mathsf{next} \mapsto 1]$	1	3	*2*	*3*	*1*	*1*	*0*	*0*