

Semantik von Programmiersprachen – SS 2017

<http://pp.ipd.kit.edu/lehre/SS2017/semantik>

Lösungen zu Blatt 6: Erweiterungen zu While

Besprechung: 06.06.2017

1. Welche der folgenden Aussagen sind richtig, welche falsch? (H)

- (a) $c_1 \text{ or } c_2$ und $c_2 \text{ or } c_1$ sind äquivalent bzgl. der Big-Step-Semantik.
- (b) $c_1 \text{ or } c_2$ und $c_2 \text{ or } c_1$ sind äquivalent bzgl. der Small-Step-Semantik.
- (c) $x := 0; y := 0; \text{ while } (y == 0) \text{ do } (x := x + 1 \text{ or } y := 1)$ terminiert immer.
- (d) $(\text{while } (b) \text{ do } c_1) \text{ or } (\text{while } (b) \text{ do } c_2)$ und $\text{while } (b) \text{ do } (c_1 \text{ or } c_2)$ sind äquivalent bzgl. der Big-Step-Semantik.
- (e) $x := 5 \text{ or } x := 6$ und $x := 5 \parallel x := 6$ sind semantisch äquivalent.
- (f) $c_1 \parallel (c_2 \parallel c_3) = (c_1 \parallel c_2) \parallel c_3$
- (g) $c_1 \parallel c_2$ und $c_2 \parallel c_1$ sind äquivalent bzgl. der Small-Step-Semantik.
- (h) Die Big-Step-Semantik von While_B ist nicht deterministisch.
- (i) Nach Ausführung von $\{ \text{var } x = 1; y := x + 1; \{ \text{var } y = 3; x := y + 2; \{ \text{var } x = 6; z := x + y \}; y := z \}; z := x + y + z \}$ hat z den Wert 24.
- (j) $\{ \text{var } z = 142; \{ \text{var } x = x + 1; z := x \}; x := z - 1 \}$ ist semantisch äquivalent zu `skip`.

Lösung:

- (1a) Richtig. Es genügt, zu zeigen: Wenn $\langle c_1 \text{ or } c_2, \sigma \rangle \Downarrow \sigma'$, dann $\langle c_2 \text{ or } c_1, \sigma \rangle \Downarrow \sigma'$. Durch Regelinversion der Annahme ergeben sich zwei Fälle:
 - Fall OR1_{BS}: $\langle c_1, \sigma \rangle \Downarrow \sigma'$. Mit Regel OR2_{BS} folgt $\langle c_2 \text{ or } c_1, \sigma \rangle \Downarrow \sigma'$.
 - Fall OR2_{BS}: Analog. □
- (1b) Richtig. Es genügt, zu zeigen: Wenn $\langle c_1 \text{ or } c_2, \sigma \rangle \rightarrow_1 \langle c, \sigma' \rangle$, dann $\langle c_2 \text{ or } c_1, \sigma \rangle \rightarrow_1 \langle c, \sigma' \rangle$. Durch Regelinversion der Annahme ergeben sich zwei Fälle:
 - Fall OR1_{SS}: $\langle c_1 \text{ or } c_2, \sigma \rangle \rightarrow_1 \langle c_1, \sigma \rangle$. Mit Regel OR2_{SS} folgt $\langle c_2 \text{ or } c_1, \sigma \rangle \rightarrow_1 \langle c_1, \sigma \rangle$.
 - Fall OR2_{SS}: Analog. □
- (1c) Falsch. In der Big-Step-Semantik hat das Programm für jeden Anfangszustand eine Ableitung, aber über Termination kann man bei nichtdeterministischen Sprachen bzw. Semantiken nur mit der Small-Step-Semantik sinnvoll reden. Wenn immer der linke Teil des Schleifenrumpfes gewählt wird, terminiert die Schleife nicht. Der `or`-Operator ist nicht fair, er kann unendlich oft gleich entscheiden – gleiches gilt auch für den Paralleloperator. Dieses Programm generiert übrigens eine zufällige, positive Zahl.
- (1d) Falsch. In $P_2 = \text{while } (b) \text{ do } (c_1 \text{ or } c_2)$ entscheidet sich bei jedem Schleifendurchlauf neu, ob c_1 oder c_2 ausgeführt wird; bei $P_1 = (\text{while } (b) \text{ do } c_1) \text{ or } (\text{while } (b) \text{ do } c_2)$ fällt diese Entscheidung einmalig beim Programmstart.

Für $b = x \leq 3$, $c_1 = x := x + 1$, $c_2 = x := x * 2$ und Anfangszustand $\sigma = [x \mapsto 2]$ gilt $\langle P_2, \sigma \rangle \Downarrow \sigma[x \mapsto 6]$, aber nicht $\langle P_1, \sigma \rangle \Downarrow \sigma[x \mapsto 6]$.

Schleifen sind das syntaktische Konstrukt in While_{ND} , das nicht mit $_ \text{ or } _$ distribuiert, da alle anderen die nichtdeterministische Fallunterscheidung nur einmal ausführen und es irrelevant ist, ob dies bereits am Anfang geschieht oder später. Der Paralleloperator $_ \parallel _$ distribuiert übrigens auch nicht mit $_ \text{ or } _$.

- (1e) Richtig. Für beide Programme sind die möglichen Endzustände die, die x den Wert 5 oder 6 zuweisen. Im Allgemeinen sind $c_1 \text{ or } c_2$ und $c_1 \parallel c_2$ aber nicht semantisch äquivalent: Bei $c_1 \text{ or } c_2$ wird *entweder* c_1 *oder* c_2 ausgeführt, bei $c_1 \parallel c_2$ werden immer c_1 *und* c_2 ausgeführt, nur die Reihenfolge und Verzahnung ist nichtdeterministisch.
- (1f) Falsch. Syntaktisch beschreiben die beiden Programme verschiedene Syntaxbäume. Allerdings sind beide äquivalent in der Small-Step-Semantik. Zum Beweis verwendet man die Bisimulation

$$B \equiv \{(c_1 \parallel (c_2 \parallel c_3), (c_1 \parallel c_2) \parallel c_3) \mid c_1, c_2, c_3 \in \text{Com}\} \cup \{(c, c) \mid c \in \text{Com}\}$$

mit $(c_1 \parallel (c_2 \parallel c_3), (c_1 \parallel c_2) \parallel c_3) \in B$. Dann zeigt man für $(c, c') \in B$: Wenn $\langle c, \sigma \rangle \rightarrow_1 \langle c^*, \sigma' \rangle$, dann gibt es ein c'' mit $\langle c', \sigma \rangle \rightarrow_1 \langle c'', \sigma' \rangle$ und $(c^*, c'') \in B$, und umgekehrt.

Bemerkung: Wenn man die kombinierte Regel PARSKIP statt der Regeln PARSKIP1 und PARSKIP2 verwenden würde, wäre die Bisimulation wesentlich komplizierter, da PARSKIP z. B. zwar in $c_1 \parallel (\text{skip} \parallel \text{skip})$ anwendbar ist, in $(c_1 \parallel \text{skip}) \parallel \text{skip}$ für $c_1 \neq \text{skip}$ erst später anwendbar sein wird, wenn c_1 vollständig ausgewertet ist. Für alle Zwischenkonfigurationen muss dieser Versatz in der Bisimulation abgebildet werden.

- (1g) Richtig. Beweis: Bisimulationsprinzip wie bei (1f). Bisimulationsrelation:

$$B \equiv \{(c_1 \parallel c_2, c_2 \parallel c_1) \mid c_1, c_2 \in \text{Com}\} \cup \{(c, c) \mid c \in \text{Com}\}$$

- (1h) Falsch.

- (1i) Falsch. z hat den Wert 16.

- (1j) Richtig. Beweis in der Big-Step-Semantik mittels eines Ableitungsbaums:

$$\frac{\frac{\langle z := x, \sigma[z \mapsto 142, x \mapsto \sigma(x) + 1] \rangle \Downarrow \sigma[z \mapsto \sigma(x) + 1, x \mapsto \sigma(x) + 1]}{\langle \{ \text{var } x = x + 1; z := x \}, \sigma[z \mapsto 142] \rangle \Downarrow \sigma[z \mapsto \sigma(x) + 1, x \mapsto (\sigma[z \mapsto 142])(x)]}}{\frac{\langle x := z - 1, \sigma[z \mapsto \sigma(x) + 1] \rangle \Downarrow \sigma[z \mapsto \sigma(x) + 1, x \mapsto \sigma(x)]}{\langle \{ \text{var } x = x + 1; z := x \}; x := z - 1, \sigma[z \mapsto 142] \rangle \Downarrow \sigma[z \mapsto \sigma(x) + 1, x \mapsto \sigma(x)]}}{\langle \{ \text{var } z = 142; \{ \text{var } x = x + 1; z := x \}; x := z - 1 \}, \sigma \rangle \Downarrow \sigma[z \mapsto \sigma(x) + 1, x \mapsto \sigma(x), z \mapsto \sigma(z)]}$$

und $\sigma[z \mapsto \sigma(x) + 1, x \mapsto \sigma(x), z \mapsto \sigma(z)] = \sigma$.

2. Blöcke und Parallelität (H)

In dieser Aufgabe seien die Erweiterungen zur Parallelität While_{PAR} und zu lokalen Variablen mittels Blöcken While_B kombiniert. Was sind die möglichen Endzustände des folgenden Programms in der kombinierten Small-Step-Semantik für den Anfangszustand $[x \mapsto 1]$?

$\{ \text{var } y = 1; x := x + 1; y := y + 1; x := x + 2; y := y + 2; z := y \} \parallel$
 $\{ \text{var } y = 1; x := x * 3; y := y * 3; x := x * 4; y := y * 4; z := y \}$

Lösung: Die Formulierung der Small-Step-Regeln für Blöcke beinhaltet, dass lokale Variablen eines Blocks auch lokal für den „Thread“ sind, der den Block enthält. Im Programm zeigt sich dieser Unterschied bei der *globalen* Variable x und der *lokalen* Variable y . Der Endwert für die Variable x hängt nur von den Anweisungen ab, die x zuweisen, der für z nur von den restlichen Anweisungen. Deswegen genügt es, nur die Interleavings für die Variablen einzeln zu betrachten.

Interleavings für x :

<pre>x := x + 1 x := x + 2 x := x * 3 x := x * 4 Wert: 48</pre>	<pre>x := x + 1 x := x * 3 x := x + 2 x := x * 4 Wert: 32</pre>	<pre>x := x + 1 x := x * 3 x := x * 4 x := x + 2 Wert: 26</pre>
<pre> x := x * 3 x := x + 1 x := x + 2 x := x * 4 Wert: 24</pre>	<pre> x := x * 3 x := x + 1 x := x * 4 x := x + 2 Wert: 18</pre>	<pre> x := x * 3 x := x * 4 x := x + 1 x := x + 2 Wert: 15</pre>

Da y in beiden „Threads“ lokal ist, ist nur das Interleaving für die beiden Zuweisungen an z relevant. Dementsprechend hat z am Ende entweder den Wert 4 oder den Wert 12.

3. Exceptions, break und continue (H)

Exceptions wie in der Vorlesung vorgestellt, können verwendet werden, um `break` und `continue` für Schleifen zu simulieren. `break` beendet sofort die innerste umgebende Schleife, `continue` beendet den aktuellen Schleifendurchlauf und setzt mit der Prüfung der Schleifenbedingung fort.

Beschreiben Sie, wie sich `While` mit `break` und `continue` als Quellcodetransformation auf `While` mit Exceptions abbilden lässt. Wie sähe eine Implementierung von `break` mit Label aus?

Lösung: Man braucht zwei neue Exception-Namen, z. B. `break` und `continue`. Jede Schleife wird mit zwei Blöcken ergänzt:

ursprünglich	wird implementiert als
<code>while (b) do c</code>	<code>try (while (b) do try c catch continue skip) catch break skip</code>
<code>break</code>	<code>raise break</code>
<code>continue</code>	<code>raise continue</code>
<code>lbl: c</code>	<code>try c catch break_lbl skip</code>
<code>break lbl</code>	<code>raise break_lbl</code>

Beachten Sie, dass sich diese Implementierung nicht direkt mit der Implementierung von `for` mittels `while` kombinieren lässt, da auch bei `continue` die Zählvariable erhöht werden muss.

4. Goto und Small-Step-Semantik mit Continuations (Ü)

In dieser Aufgabe soll eine Small-Step-Semantik für `While` mit `goto` definiert werden. Dazu sei `Lab` eine Menge von Labels, die typischerweise mit l bezeichnet werden. Mit diesen können beliebige Stellen im Programm markiert werden, dafür erweitern wir die Syntax von `While`:

Com $c ::= l \mid \text{goto } l \mid \dots$

- (a) Unsere bisherige Small-Step-Semantik ist nicht geeignet, `goto` sauber abzubilden: Die Regel `SEQ1SS` für $c_1; c_2$ erlaubt es nicht, dass c_1 wegspringt. Daher stellen wir die Semantik auf Continuations um. Ein Zustand unserer Semantik ist nun $\langle cs, \sigma \rangle$ und besagt, dass statt einem einzelnen Programm c die Liste von Programmen cs auszuführen ist.

Geben Sie die Regeln für `While` in dieser Semantik an. Dies ist ohne rekursive Regeln wie `SEQ1SS` möglich! Was sind die blockierten Zustände?

- (b) Ergänzen Sie diese Small-Step-Semantik um Regeln für $l:$ und `goto l`. Da ein Sprung irgendwo im Programm landen kann, müssen alle Small-Step-Regeln nun auch das komplette Programm durchschleifen. Da es nicht verändert wird, schreibt man es vor die Relation: $c \vdash \langle cs_1, \sigma_1 \rangle \rightarrow_1 \langle cs_2, \sigma_2 \rangle$ besagt, dass während der Auswertung des Programms c die Programmfragmente cs_1 im Zustand σ_1 in einem Schritt zu cs_2 im Zustand σ_2 ausgewertet werden.

Für die Regel für `goto` werden Sie eine Funktion benötigen, die in einem Programm c nach dem Label l sucht und ein Programm $\mathcal{L}_l(c)$ zurückgibt, das die Ausführung von c ab dem Label l beschreibt. Definieren Sie diese Funktion. Sie können dabei das Prädikat $l \in c_1$ verwenden, das wahr ist, wenn im Programm(fragment) c_1 das Label l vorkommt. Gehen Sie davon aus, dass in jedem Programm jedes Label höchstens einmal gesetzt wurde, und beachten Sie nur Labels, die auch im Programm vorkommen.

- (c) Gegeben sei das folgende While-Programm c . Geben Sie $\mathcal{L}_{\text{lab}}(c)$ und die Ableitungsfolge von c in einem Zustand σ an.

`y := 0; (while (y == 1) do (lab;; y := 2)); if (y == 0) then goto lab else skip`

Lösung:

- 4a) Folgende Regeln implementieren die Small-Step-Continuations-Semantik, wobei wieder $c \cdot cs$ abkürzend für $[c] ++ cs$ steht.

$$\text{SEQ}_{\text{CSS}}: \langle c_0; c_1 \cdot cs, \sigma \rangle \rightarrow_1 \langle c_0 \cdot c_1 \cdot cs, \sigma \rangle \quad \text{SKIP}_{\text{CSS}}: \langle \text{skip} \cdot cs, \sigma \rangle \rightarrow_1 \langle cs, \sigma \rangle$$

$$\text{ASS}_{\text{CSS}}: \langle x := a \cdot cs, \sigma \rangle \rightarrow_1 \langle cs, \sigma[x \mapsto \mathcal{A} \llbracket a \rrbracket \sigma] \rangle$$

$$\text{IFTT}_{\text{CSS}}: \frac{\mathcal{B} \llbracket b \rrbracket \sigma = \text{tt}}{\langle \text{if } (b) \text{ then } c_0 \text{ else } c_1 \cdot cs, \sigma \rangle \rightarrow_1 \langle c_0 \cdot cs, \sigma \rangle}$$

$$\text{IFFF}_{\text{CSS}}: \frac{\mathcal{B} \llbracket b \rrbracket \sigma = \text{ff}}{\langle \text{if } (b) \text{ then } c_0 \text{ else } c_1 \cdot cs, \sigma \rangle \rightarrow_1 \langle c_1 \cdot cs, \sigma \rangle}$$

$\text{WHILE}_{\text{CSS}}:$

$$\langle \text{while } (b) \text{ do } c \cdot cs, \sigma \rangle \rightarrow_1 \langle \text{if } (b) \text{ then } c; \text{ while } (b) \text{ do } c \text{ else skip} \cdot cs, \sigma \rangle$$

Die einzigen blockierenden Zustände sind $\langle [], \sigma \rangle$.

- 4b) Die Funktion \mathcal{L} definieren wir rekursiv über c . Dabei überlegen wir uns jeweils, wie das c in der Small-Step-Relation in dem Moment aussehen würde, wenn wir auf normalem Wege an dem Label angekommen wären. Da wir nur Labels betrachten, die im Programm vorkommen, müssen wir keine Fälle für `skip`, `goto` und die Zuweisung angeben:

$$\mathcal{L}_l(l:) := l:$$

$$\mathcal{L}_l(c_1; c_2) := \begin{cases} \mathcal{L}_l(c_1); c_2 & \text{falls } l \in c_1 \\ \mathcal{L}_l(c_2) & \text{falls } l \in c_2 \end{cases}$$

$$\mathcal{L}_l(\text{if } (b) \text{ then } c_1 \text{ else } c_2) := \begin{cases} \mathcal{L}_l(c_1) & \text{falls } l \in c_1 \\ \mathcal{L}_l(c_2) & \text{falls } l \in c_2 \end{cases}$$

$$\mathcal{L}_l(\text{while } (b) \text{ do } c) := \mathcal{L}_l(c); \text{ while } (b) \text{ do } c$$

Nun können wir die zusätzlichen Small-Step-Regeln angeben. Zu beachten ist, dass die Regel GOTO_{CSS} die bisherigen Continuations verwirft; das ist der Grund für den Umstieg auf eine Small-Step-Semantik mit Continuations!

$$\text{LABEL}_{\text{CSS}}: c \vdash \langle l: \cdot cs, \sigma \rangle \rightarrow_1 \langle cs, \sigma \rangle$$

$$\text{GOTO}_{\text{CSS}}: c \vdash \langle \text{goto } l \cdot cs, \sigma \rangle \rightarrow_1 \langle [\mathcal{L}_l(c)], \sigma \rangle$$

- 4c) Für das in der Aufgabenstellung gegebene Programm c ist

$$\mathcal{L}_{\text{lab}}(c) = \text{lab}; y := 2; c_w; c_i,$$

wobei wir die Abkürzungen $c_w := \text{while } (y == 1) \text{ do } (\text{lab}; y := 2)$ und $c_i := \text{if } (y == 0) \text{ then goto lab else skip}$ verwenden.

Damit ergibt sich folgende Ableitungsfolge für c , wobei $\sigma_i := \sigma[y \mapsto i]$ und wir davon ausgehen, dass $;$ links-assoziativ ist:

$$\begin{aligned}
c \vdash \langle [c], \sigma \rangle &\rightarrow_1 \langle [y := 0; c_w, c_i], \sigma \rangle \\
&\rightarrow_1 \langle [y := 0, c_w, c_i], \sigma \rangle \\
&\rightarrow_1 \langle [c_w, c_i], \sigma_0 \rangle \\
&\rightarrow_1 \langle [\text{if } (y == 1) \text{ then lab}; y := 2; c_w \text{ else skip}, c_i], \sigma_0 \rangle \\
&\rightarrow_1 \langle [\text{skip}, c_i], \sigma_0 \rangle \\
&\rightarrow_1 \langle [c_i], \sigma_0 \rangle \\
&\rightarrow_1 \langle [\text{goto lab}], \sigma_0 \rangle \\
&\rightarrow_1 \langle [\mathcal{L}_{\text{lab}}(c)], \sigma_0 \rangle \\
&\rightarrow_1 \langle [\text{lab}; y := 2; c_w, c_i], \sigma_0 \rangle \\
&\rightarrow_1 \langle [\text{lab}; y := 2, c_w, c_i], \sigma_0 \rangle \\
&\rightarrow_1 \langle [\text{lab}; y := 2, c_w, c_i], \sigma_0 \rangle \\
&\rightarrow_1 \langle [y := 2, c_w, c_i], \sigma_0 \rangle \\
&\rightarrow_1 \langle [c_w, c_i], \sigma_2 \rangle \\
&\rightarrow_1 \langle [\text{if } (y == 1) \text{ then lab}; y := 2; c_w \text{ else skip}, c_i], \sigma_2 \rangle \\
&\rightarrow_1 \langle [\text{skip}, c_i], \sigma_2 \rangle \\
&\rightarrow_1 \langle [c_i], \sigma_2 \rangle \\
&\rightarrow_1 \langle [], \sigma_2 \rangle
\end{aligned}$$