Karlsruher Institut für Technologie

Lehrstuhl für Programmierparadigmen
Sebastian Ullrich sebastian.ullrich@kit.edu
Max Wagner maximilian.wagner@kit.edu

Semantik von Programmiersprachen – SS 2019

http://pp.ipd.kit.edu/lehre/SS2019/semantik

Lösungen zu Blatt 6: Erweiterungen zu While Besprechung: 03.06.2019

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

(a) c_1 `or` c_2 und c_2 `or` c_1 sind äquivalent bzgl. der Big-Step-Semantik.

(b) c_1 `or` c_2 und c_2 `or` c_1 sind äquivalent bzgl. der Small-Step-Semantik.

(c) `x := 0; y := 0; while (y == 0) do (x := x + 1 or y := 1)` terminiert immer.

(d) `(while (`b`) do` c_1`) or (while (`b`) do` c_2`)` und `while (`b`) do (`c_1 `or` c_2`)` sind äquivalent bzgl. der Big-Step-Semantik.

(e) `x := 5 or x := 6` und `x := 5 || x := 6` sind semantisch äquivalent.

(f) c_1 `||` $(c_2$ `||` $c_3) = (c_1$ `||` $c_2)$ `||` c_3

(g) c_1 `||` c_2 und c_2 `||` c_1 sind äquivalent bzgl. der Small-Step-Semantik.

(h) Die Big-Step-Semantik von While_B ist nicht deterministisch.

(i) Nach Aufführung von `{ var x = 1; y := x + 1; { var y = 3; x := y + 2; { var x = 6; z := x + y }; y := z }; z := x + y + z }` hat `z` den Wert 24.

(j) `{ var z = 142; { var x = x + 1; z := x }; x := z - 1 }` ist semantisch äquivalent zu `skip`.

Lösung:

(1a) Richtig. Es genügt, zu zeigen: Wenn $\langle c_1$ `or` $c_2,\ \sigma\rangle \Downarrow \sigma'$, dann $\langle c_2$ `or` $c_1,\ \sigma\rangle \Downarrow \sigma'$. Durch Regelinversion der Annahme ergeben sich zwei Fälle:
- Fall $\textsc{Or1}_{\text{BS}}$: $\langle c_1,\ \sigma\rangle \Downarrow \sigma'$. Mit Regel $\textsc{Or2}_{\text{BS}}$ folgt $\langle c_2$ `or` $c_1,\ \sigma\rangle \Downarrow \sigma'$.
- Fall $\textsc{Or2}_{\text{BS}}$: Analog. □

(1b) Richtig. Es genügt, zu zeigen: Wenn $\langle c_1$ `or` $c_2,\ \sigma\rangle \to_1 \langle c,\ \sigma'\rangle$, dann $\langle c_2$ `or` $c_1,\ \sigma\rangle \to_1 \langle c,\ \sigma'\rangle$. Durch Regelinversion der Annahme ergeben sich zwei Fälle:
- Fall $\textsc{Or1}_{\text{SS}}$: $\langle c_1$ `or` $c_2,\ \sigma\rangle \to_1 \langle c_1,\ \sigma\rangle$. Mit Regel $\textsc{Or2}_{\text{SS}}$ folgt $\langle c_2$ `or` $c_1,\ \sigma\rangle \to_1 \langle c_1,\ \sigma\rangle$.
- Fall $\textsc{Or2}_{\text{SS}}$: 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 Parallelоperator.

Dieses Programm generiert übrigens eine zufällige, positive Zahl.

(1d) Falsch. In $P_2 =$ `while (`b`) do (`c_1 `or` c_2`)` entscheidet sich bei jedem Schleifendurchlauf neu, ob c_1 oder c_2 ausgeführt wird; bei $P_1 =$ `(while (`b`) do` c_1`) or (while (`b`) do` c_2`)` fällt diese Entscheidung einmalig beim Programmstart.

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

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

(1e) Richtig. Für beide Programme sind die möglichen Endzustände die, die `x` den Wert 5 oder 6 zuweisen. Im Allgemeinen sind c_1 `or` c_2 und c_1 `||` c_2 aber nicht semantisch äquivalent: Bei c_1 `or` c_2 wird *entweder* c_1 *oder* c_2 ausgeführt, bei c_1 `||` 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

$$B \equiv \{(c_1\ \texttt{||}\ (c_2\ \texttt{||}\ c_3), (c_1\ \texttt{||}\ c_2)\ \texttt{||}\ c_3) \mid c_1, c_2, c_3 \in \mathsf{Com}\} \cup \{(c, c) \mid c \in \mathsf{Com}\}$$

Dann zeigt man für $(c, c') \in B$: Wenn $\langle c,\ \sigma\rangle \to_1 \langle c^*,\ \sigma'\rangle$, dann gibt es ein c'' mit $\langle c',\ \sigma\rangle \to_1 \langle c'',\ \sigma'\rangle$ und $(c^*, c'') \in B$, und umgekehrt. Mit dieser Aussage hat man dann, dass Programme, die miteinander B-verwandt sind, einander semantisch äquivalent simulieren können, ohne dabei diese Relation zu verlassen. Dann braucht es noch ein induktives Argument dafür, wieso eine beliebige (potenziell unendliche) Ausführungsfolge für ein c in eine gültige Ausführungsfolge für ein c' transformiert werden kann (unter der Vorraussetzung, dass $(c, c') \in B$).

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 `||` `(skip || skip)` anwendbar ist, in `(`c_1 `||` `skip) || skip` für $c_1 \neq$ `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:

$$B \equiv \{(c_1\ \texttt{||}\ c_2, c_2\ \texttt{||}\ c_1) \mid c_1, c_2 \in \mathsf{Com}\} \cup \{(c, c) \mid c \in \mathsf{Com}\}$$

(1h) Falsch.

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

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

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

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

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 $[\texttt{x} \mapsto 1]$?

```
({ var y = 1; x := x + 1; y := y + 1; x := x + 2; y := y + 2; z := y }) ||
({ var y = 1; x := x * 3; y := y * 3; 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`:

`x := x + 1`	`x := x + 1`	`x := x * 3`
`x := x + 2`	`x := x * 3`	`x := x + 1`
`x := x * 3`	`x := x + 2`	`x := x + 2`
Wert: 12	Wert: 8	Wert: 6

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**
`while (b) do c`	`try (while (b) do try c catch continue skip) catch break skip`
`break`	`raise break`
`continue`	`raise continue`
lbl`: c`	`try c catch break_`*lbl* `skip`
`break` *lbl*	`raise break_`*lbl*

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`:` | `goto` l | ...

(a) Unsere bisherige Small-Step-Semantik ist nicht geeignet, `goto` sauber abzubilden: Die Regel $\textsc{Seq1}_{\text{SS}}$ 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 $\textsc{Seq1}_{\text{SS}}$ 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 Program landen kann, müssen alle Small-Step-Regeln nun auch das komplette Programm durchschleifen. Da es nicht verändert wird, schreibt man es vor die Relation: $c \vdash \langle cs_1,\ \sigma_1\rangle \to_1 \langle cs_2,\ \sigma_2\rangle$ 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 $\mathcal{L}_{\texttt{lab}}(c)$ 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.

$$\textsc{Seq}_{\text{CSS}}\colon \langle c_1\texttt{;}\ c_2 \cdot cs,\ \sigma\rangle \to_1 \langle c_1 \cdot c_2 \cdot cs,\ \sigma\rangle \qquad \textsc{Skip}_{\text{CSS}}\colon \langle \texttt{skip} \cdot cs,\ \sigma\rangle \to_1 \langle cs,\ \sigma\rangle$$

$$\textsc{Ass}_{\text{CSS}}\colon \langle x\ \texttt{:=}\ a \cdot cs,\ \sigma\rangle \to_1 \langle cs,\ \sigma[x \mapsto \mathcal{A}[\![a]\!]\,\sigma]\rangle$$

$$\textsc{IfTT}_{\text{CSS}}\colon \frac{\mathcal{B}[\![b]\!]\,\sigma = \mathbf{tt}}{\langle \texttt{if (}b\texttt{) then }c_1\texttt{ else }c_2 \cdot cs,\ \sigma\rangle \to_1 \langle c_1 \cdot cs,\ \sigma\rangle}$$

$$\textsc{IfFF}_{\text{CSS}}\colon \frac{\mathcal{B}[\![b]\!]\,\sigma = \mathbf{ff}}{\langle \texttt{if (}b\texttt{) then }c_1\texttt{ else }c_2 \cdot cs,\ \sigma\rangle \to_1 \langle c_2 \cdot cs,\ \sigma\rangle}$$

$$\textsc{While}_{\text{CSS}}\colon \langle \texttt{while (}b\texttt{) do }c \cdot cs,\ \sigma\rangle \to_1 \langle \texttt{if (}b\texttt{) then }c\texttt{; while (}b\texttt{) do }c\texttt{ else skip} \cdot cs,\ \sigma\rangle$$

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:

$$\mathcal{L}_l(l\texttt{:}) \coloneqq l\texttt{:}$$

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

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

$$\mathcal{L}_l(\texttt{while (}b\texttt{) do }c) \coloneqq \mathcal{L}_l(c)\texttt{; while (}b\texttt{) do }c$$

Nun können wir die zusätzlichen Small-Step-Regeln angeben. Die Continuations geben uns eine lineare Repräsentation des Residualprogramms (statt des baumförmigen Sequenz-Konstruktes). Die eindeutige Aufteilung in aktuelles Statement und vollständiges Restprogramm erlaubt es uns, in $\textsc{Goto}_{\text{CSS}}$ das *komplette* Residualprogramm zu verwerfen; ohne Continuations ist das Restprogramm syntaktisch gar nicht greifbar!

$$\textsc{Label}_{\text{CSS}}\colon c \vdash \langle l\texttt{:} \cdot cs,\ \sigma\rangle \to_1 \langle cs,\ \sigma\rangle$$

$$\textsc{Goto}_{\text{CSS}}\colon c \vdash \langle \texttt{goto }l \cdot cs,\ \sigma\rangle \to_1 \langle [\mathcal{L}_l(c)],\ \sigma\rangle$$

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

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

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

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

$$
\begin{aligned}
c \vdash \langle [c],\ \sigma\rangle &\to_1 \langle [\texttt{y := 0; } c_w, c_i],\ \sigma\rangle \\
&\to_1 \langle [\texttt{y := 0}, c_w, c_i],\ \sigma\rangle \\
&\to_1 \langle [c_w, c_i],\ \sigma_0\rangle \\
&\to_1 \langle [\texttt{if (y == 1) then lab:; y := 2; } c_w \texttt{ else skip}, c_i],\ \sigma_0\rangle \\
&\to_1 \langle [\texttt{skip}, c_i],\ \sigma_0\rangle \\
&\to_1 \langle [c_i],\ \sigma_0\rangle \\
&\to_1 \langle [\texttt{goto lab}],\ \sigma_0\rangle \\
&\to_1 \langle [\mathcal{L}_{\texttt{lab}}(c)],\ \sigma_0\rangle \\
&\to_1 \langle [\texttt{lab:; y := 2; } c_w, c_i],\ \sigma_0\rangle \\
&\to_1 \langle [\texttt{lab:; y := 2}, c_w, c_i],\ \sigma_0\rangle \\
&\to_1 \langle [\texttt{lab:}, \texttt{y := 2}, c_w, c_i],\ \sigma_0\rangle \\
&\to_1 \langle [\texttt{y := 2}, c_w, c_i],\ \sigma_0\rangle \\
&\to_1 \langle [c_w, c_i],\ \sigma_2\rangle \\
&\to_1 \langle [\texttt{if (y == 1) then lab:; y := 2; } c_w \texttt{ else skip}, c_i],\ \sigma_2\rangle \\
&\to_1 \langle [\texttt{skip}, c_i],\ \sigma_2\rangle \\
&\to_1 \langle [c_i],\ \sigma_2\rangle \\
&\to_1 \langle [],\ \sigma_2\rangle
\end{aligned}
$$