Karlsruher Institut für Technologie

Lehrstuhl für Programmierparadigmen

Denis Lohner denis.lohner@kit.edu

Sebastian Ullrich sebastian.ullrich@kit.edu

Semantik von Programmiersprachen – SS 2017

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

Lösungen zu Blatt 2: Big-Step-Semantik Besprechung: 08.05.2017

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

(a) `false` = **ff**

(b) `skip; x := true` ist ein While-Programm.

(c) `x := y * z` ≠ `x := z * y`.

(d) `x + y` = `x - (0 - y)`.

(e) `x := y + 1; y := x` und `y := x; x := y + 1` haben das gleiche Verhalten.

(f) `if (not` b`) then` c_0 `else` c_1 und `if (`b`) then` c_1 `else` c_0 haben das gleiche Verhalten.

(g) Es gibt a und σ, für die $\mathcal{A}[\![a]\!]\,\sigma$ nicht definiert ist.

(h) Es gibt keine σ und σ', so dass $\langle$`while (true) do skip`$,\ \sigma\rangle \Downarrow \sigma'$.

(i) Ersetzt man die Regeln IfTT$_{\text{BS}}$ und IfFF$_{\text{BS}}$ durch folgende kombinierte Regel, ändert sich die Big-Step-Semantik nicht:

$$\frac{\langle c_0,\ \sigma\rangle \Downarrow \sigma' \qquad \langle c_1,\ \sigma\rangle \Downarrow \sigma'' \qquad \mathcal{B}[\![b]\!]\,\sigma = \mathbf{tt} \longrightarrow \sigma''' = \sigma' \qquad \mathcal{B}[\![b]\!]\,\sigma = \mathbf{ff} \longrightarrow \sigma''' = \sigma''}{\langle \texttt{if (}b\texttt{) then } c_0 \texttt{ else } c_1,\ \sigma\rangle \Downarrow \sigma'''}$$

Lösung:

(1a) Falsch. `false` ist ein syntaktisches Konstrukt und **ff** ist ein Wahrheitswert. Es gilt: $\mathcal{B}[\![\texttt{false}]\!]\,\sigma = \mathbf{ff}$.

(1b) Falsch. `true` ist kein arithmetischer Ausdruck, wie von der Grammatik für Zuweisungen gefordert.

(1c) Richtig. Das syntaktische `*` ist nicht kommutativ.

(1d) Richtig. `+` ist nur syntaktischer Zucker.

(1e) Falsch. Beispiel: $\sigma = [\texttt{x} \mapsto 7,\ \texttt{y} \mapsto 9]$.
$\langle$`x := y + 1; y := x`$,\ \sigma\rangle \Downarrow [\texttt{x} \mapsto 10,\ \texttt{y} \mapsto 10]$
$\langle$`y := x; x := y + 1`$,\ \sigma\rangle \Downarrow [\texttt{x} \mapsto 8,\ \texttt{y} \mapsto 7]$

(1f) Richtig.
Sei I_1 = `if (not` b`) then` c_0 `else` c_1, I_2 = `if (`b`) then` c_1 `else` c_0.
Zu zeigen: $\langle I_1,\ \sigma\rangle \Downarrow \sigma'$ gdw. $\langle I_2,\ \sigma\rangle \Downarrow \sigma'$.
Fall „⇒“: Sei $\langle I_1,\ \sigma\rangle \Downarrow \sigma'$. Dann ist entweder $\mathcal{B}[\![\texttt{not } b]\!]\,\sigma = \texttt{true}$ und $\langle c_0,\ \sigma\rangle \Downarrow \sigma'$ oder $\mathcal{B}[\![\texttt{not } b]\!]\,\sigma = \texttt{false}$ und $\langle c_1,\ \sigma\rangle \Downarrow \sigma'$. Im ersten Fall ist $\mathcal{B}[\![b]\!]\,\sigma = \texttt{false}$ und damit $\langle I_2,\ \sigma\rangle \Downarrow \sigma'$. Im zweiten Fall ist $\mathcal{B}[\![b]\!]\,\sigma = \texttt{true}$ und damit ebenfalls $\langle I_2,\ \sigma\rangle \Downarrow \sigma'$.
Fall „⇐“: Analog.

(1g) Falsch. $\mathcal{A}[\![\_]\!]\,\_$ ist strukturell rekursiv für alle arithmetischen Ausdrücke definiert, auch für alle Variablen, da Zustände per Definition *jeder* Variablen aus Var einen Wert zuweisen.

(1h) Richtig. Eine induktive Definition enthält keine unendlichen Ableitungsbäume, die man hier bräuchte. Sonst wäre auch die Induktionsregel falsch: Angenommen, es gibt Zustände σ und σ', so dass $\langle$`while (true) do skip`, $\sigma\rangle \Downarrow \sigma'$ gilt. Dann kann man das Prädikat $P(c, \sigma, \sigma') \equiv c \neq$ `while (true) do skip` für $c =$ `while (true) do skip` und eben jene σ, σ' zeigen, was natürlich Unsinn ist.

(1i) Falsch. Gegenbeispiel: $b =$ `true` und $c_0 =$ `skip` und $c_1 =$ `while (true) do skip`. Die neue Regel verlangt, dass beide Zweige terminieren; die alten verlangen nur die Termination des Zweigs, der ausgewählt wird.

2. Big-Step-Semantik in Prolog (H)

Implementieren Sie die Big-Step-Semantik für While in Prolog. Ein While-Programm wird in der folgenden Syntax eingegeben: Für das syntaktisch repräsentative While-Programm

```
n := 42; while (true && not (n <= 1)) do if (n <= 1) then skip else n := n - (1 * 23)
```

schreiben wir in Prolog

```
n:=42; while(and(true, not(n <= 1)), cond(n <= 1, skp, n := n − (1∗23))).
```

Die Datei `common.pl` auf der Webseite zur Übung definiert dazu den Operator <= und stellt unter anderem folgende Prädikate bereit, die Sie natürlich benutzen sollen (Sie importieren die Definition mit der Prolog-Klausel :− consult('common.pl'). in Ihre Datei):

- evalA(S,A,V) ist wahr, wenn der arithmetische Ausdruck A, im Zustand S ausgewertet, den Integer-Wert V ergibt.
- evalB(S,B,V) ist wahr, wenn der Bool'sche Ausdruck B, im Zustand S ausgewertet, den Bool'schen Wert V ergibt, wobei Bool'sche Werte entwender tt oder ff sind.
- set(S1,Var,Val,S2) ist wahr, wenn sich der Zustand S2 aus dem Zustand S1 ergibt, in dem die Variable Var auf den Wert Val gesetzt wird.
- get(S,Var,Val) ist wahr, wenn im Zustand S die Variable Var den Wert Val hat. Wenn die Variable nicht gesetzt ist, wird ein beliebiger[1] Wert zurückgegeben, da wir mit totalen Zustandsabbildungen arbeiten wollen.

Variablen sind Prolog-Atome, Zustände werden als Listen von Paaren dargestellt; [] ist der initiale Zustand, [n−3,m−4] entspricht $[\mathtt{n} \mapsto 3, \mathtt{m} \mapsto 4]$. Sie sollen nun ein Prolog-Prädikat evalBS(Prog, S1, S2) definieren, das genau dann erfüllt ist, wenn Prog dem Programm c, S1 dem Zustand σ_1 und S2 dem Zustand σ_2 entspricht und $\langle c,\ \sigma_1\rangle \Downarrow \sigma_2$ gilt.

Schreiben Sie folgendes While-Programm als Prolog-Term und werten Sie es aus. Was berechnet es, in Abhängigkeit von dem Initialwert von `n`?

```
m := 1; while (1 <= n) do m := m * n; n := n - 1
```

Lösung: Eine mögliche Implementierung ist die Folgende:

```
:− consult(common).

evalBS(skp, S, S).
evalBS((P1;P2), S1, S3) :−
        evalBS(P1, S1, S2),
        evalBS(P2, S2, S3).
evalBS(Var := Aexp, S1, S2) :−
        evalA(S1, Aexp, Value),
        set(S1, Var, Value, S2).
```

[1] Für einen Informatiker-kompatiblen Wert von beliebig.

```
evalBS(cond(Bexp, P1, _), S1, S2) :−
        evalB(S1, Bexp, tt ),
        evalBS(P1, S1, S2).
evalBS(cond(Bexp, _, P2), S1, S2) :−
        evalB(S1, Bexp, ff ),
        evalBS(P2, S1, S2).
evalBS(while(Bexp,_), S1, S1) :−
        evalB(S1, Bexp, ff ).
evalBS(while(Bexp,P), S1, S3) :−
        evalB(S1, Bexp, tt ),
        evalBS(P, S1, S2),
        evalBS(while(Bexp,P), S2, S3).
```

Das angegebene Programm berechnet die Fakultät von `n`, wie man an der folgenden Ausführung sieht:

```
?− [bigstep ].
% common compiled 0.00 sec, 7,016 bytes
% bigstep compiled 0.00 sec, 33,496 bytes
true .

?− evalBS(m:=1; while(1 <= n, (m:=m∗n; n:=n−1)), [n−10], X).
X = [m−3628800, n−0] .
```

3. Ableitungsbäume (H)

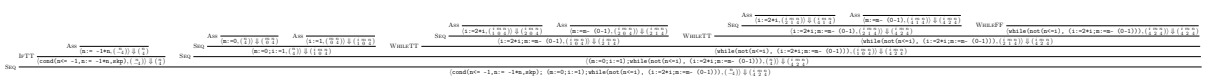
Schreiben Sie ein Programm in While, das für eine Zahl n die kleinste Zahl m bestimmt, so dass $2^m \geq |n|$, wobei $|n|$ der Absolutbetrag von n ist.

Zeichnen Sie den Ableitungsbaum einer Ausführung des Programms gemäß der Big-Step-Semantik mit $n = -4$ im Startzustand. Hören Sie auf wenn der Baum fertig ist, oder Sie überzeugt sind, dass (a) Sie das Anwenden der induktiven Regeln jetzt können und (b) dass der Computer diese Aufgabe besser können müsste.

Lösung:

Eine Lösung ist: `((if (n <= -1) then n := -1 * n else skip); (m := 0; i := 1)); while (not (n <= i)) do (i := 2 * i; m := m + 1)`

Den zugehörigen Ableitungsbaum kann man sich mit Hilfe der Prolog-Datei `derivTree.pl` von der Übungs-Homepage generieren lassen (von dem man hier nur mit einem PDF-Program mit gutem Zoom etwas hat):



4. Zählschleife (Ü)

Viele Programmiersprachen bieten neben einer `while`-Schleife auch eine `for`-Schleife an. Erweitern Sie die Syntax und Big-Step-Semantik von While um eine Zählschleife mit der Syntax `for` x `=` a `to` a' `do` c. Verfolgen Sie dabei verschiedene semantische Modellierungen. Finden Sie Programme, die je nach semantischer Variante unterschiedliches Verhalten zeigen. Hilfreiche Überlegungen: Könnte man Ihre Zählschleife auch als syntaktischen Zucker mittels `while` definieren? Terminieren alle `for`-Schleife immer? Was ist der Wert des Schleifenzählers nach Ende der Schleife? Implementieren Sie die Semantiken in Prolog und experimentieren Sie damit!

Lösung:

Wir erweitern die Syntax um eine zusätzliche Produktion für Com:
$c ::=$ `for` x `=` a `to` a' `do` c
und experimentieren mit verschiedenen Semantiken:

(a) *Darstellung mittels* `while`

Regel:

$$\textsc{For}: \frac{\langle x \texttt{ := } a\texttt{; while (x <= } a'\texttt{) do (}c\texttt{; } x \texttt{ := } x \texttt{ + 1), } \sigma\rangle \Downarrow \sigma'}{\langle \texttt{for } x \texttt{ = } a \texttt{ to } a' \texttt{ do } c, \ \sigma\rangle \Downarrow \sigma'}$$

Prolog:

```
evalBS(for(X, Y, Z, P), S1, S2) :-
        evalBS((X:=Y; while(X <= Z, (P; X := X - -1))), S1, S2).
```

Modellierungsgedanken: Die Schleife ist offensichtlich nur syntaktischer Zucker, keine Erweiterung der Sprache. Sie ist keine Zählschleife im klassischen Sinn, da die Anzahl der Ausführungen und die Werte der Zählevariable nicht schon zu Beginn der Schleife feststehen. Sie entspricht aber den `for`-Schleifen in vielen Programmiersprachen

(b) *Einfache Variante, keine echte Zählschleife*

Regeln:

$$\textsc{ForFF}: \frac{\mathcal{A}[\![a]\!]\,\sigma > \mathcal{A}[\![a']\!]\,\sigma}{\langle \texttt{for } x \texttt{ = } a \texttt{ to } a' \texttt{ do } c, \ \sigma\rangle \Downarrow \sigma}$$

$$\textsc{ForTT}: \frac{\mathcal{A}[\![a]\!]\,\sigma \leq \mathcal{A}[\![a']\!]\,\sigma \qquad \langle c, \ \sigma[x \mapsto \mathcal{A}[\![a]\!]\,\sigma]\rangle \Downarrow \sigma' \qquad \langle \texttt{for } x \texttt{ = } a \texttt{ + 1 to } a' \texttt{ do } c, \ \sigma'\rangle \Downarrow \sigma''}{\langle \texttt{for } x \texttt{ = } a \texttt{ to } a' \texttt{ do } c, \ \sigma\rangle \Downarrow \sigma''}$$

Prolog:

```
evalBS(for(_, Y, Z, _), S, S) :-
        evalA(S, Y, YV),
        evalA(S, Z, ZV),
        YV > ZV.

evalBS(for(X, Y, Z, P), S1, S3) :-
        evalA(S1, Y, YV),
        evalA(S1, Z, ZV),
        YV =< ZV,
        set(S1, X, YV, S1_),
        evalBS(P, S1_, S2),
        evalBS(for(X, Y - -1, Z, P), S2, S3).
```

Modellierungsgedanken: Hier werden Anfangswert und Obergrenze bei jeder Iteration neu berechnet. Falls diese von Variablen abhängen, die in der Schleife verändert werden, ist Termination nicht mehr garantiert. Wegen der dynamischen Code-Generierung (a wird zu a `+ 1`) ist dies nicht durch `while` ausdrückbar, also eine echte Erweiterung der Sprache.

(c) *Komplizierte Variante, echte Zählschleife.*

Für diese Variante brauchen wir die inverse Funktion $\mathcal{N}^{-1}[\![\_]\!] : \mathbb{Z} \mapsto \mathsf{Aexp}$ zu $\mathcal{N}[\![\_]\!]$, die aus einer ganzen Zahl ihre syntaktische Darstellung macht.

Regeln:

$$\textsc{ForFF}: \frac{\mathcal{A}[\![a]\!]\,\sigma > \mathcal{A}[\![a']\!]\,\sigma}{\langle \texttt{for } x \texttt{ = } a \texttt{ to } a' \texttt{ do } c, \ \sigma\rangle \Downarrow \sigma}$$

$$\textsc{ForTT}: \frac{\begin{array}{c} n = \mathcal{A}[\![a]\!]\,\sigma \qquad n' = \mathcal{A}[\![a']\!]\,\sigma \\ n \leq n' \qquad \langle c, \ \sigma[x \mapsto n]\rangle \Downarrow \sigma' \qquad \langle \texttt{for } x \texttt{ = } \mathcal{N}^{-1}[\![n+1]\!] \texttt{ to } \mathcal{N}^{-1}[\![n']\!] \texttt{ do } c, \ \sigma'\rangle \Downarrow \sigma'' \end{array}}{\langle \texttt{for } x \texttt{ = } a \texttt{ to } a' \texttt{ do } c, \ \sigma\rangle \Downarrow \sigma''[x \mapsto \sigma(x)]}$$

Prolog:

```
evalBS(for(_, Y, Z, _), S, S) :-
        evalA(S, Y, YV),
        evalA(S, Z, ZV),
        YV > ZV.

evalBS(for(X, Y, Z, P), S1, S4) :-
        evalA(S1, X, XV),
        evalA(S1, Y, YV),
        evalA(S1, Z, ZV),
        YV =< ZV,
        set(S1, X, YV, S1_),
        evalBS(P, S1_, S2),
        XVP1 is XV + 1,
        evalBS(for(X, XVP1, ZV, P), S2, S3),
        set(S3, X, XV, S4).
```

Modellierungsgedanken: Hier ist das Ziel, eine echte Zählschleife zu implementieren, das heißt Änderungen an der Zählvariable im Schleifenrumpf beeinflussen nicht die Zahl der Iterationen. Die dazu nötigen Werte „speichern“ wir in der syntaktischen Darstellung des Programms, um nicht den Zustand um einen Stack o.ä. erweitern zu müssen. Dafür wird $\mathcal{N}^{-1}[\![\_]\!]$ benötigt. Diese Implementierung zeigt echtes Block-Verhalten: Die Zählvariable überschreibt den Wert im Zustand nur für den Rumpf und wird am Ende der Ausführung wieder hergestellt. Deshalb ist dies auch nicht durch eine `while`-Schleife simulierbar, da dort immer eine Variable zum Speichern des alten Werts benötigt würde.

Beispielprogramm `for x = y to z do (x := x * 2; y := y + 3; z := z + 1)` und Anfangszustand $[\mathtt{x} \mapsto 0, \mathtt{y} \mapsto 1, \mathtt{z} \mapsto 7]$:

(a) Endzustand bei Konversion zu `while`: $[\mathtt{x} \mapsto 15, \mathtt{y} \mapsto 10, \mathtt{z} \mapsto 10]$

(b) Endzustand bei vereinfachter Variante: $[\mathtt{x} \mapsto 18, \mathtt{y} \mapsto 10, \mathtt{z} \mapsto 10]$

(c) Endzustand bei echter Zählschleife mit Block-Effekt: $[\mathtt{x} \mapsto 0, \mathtt{y} \mapsto 19, \mathtt{z} \mapsto 13]$