PROSEMINAR
1. TEIL

SEMANTIK

WINTERSEMESTER 2002 / 2003
KRÖGER, RAUSCHMAYE

VERFASSER:
AHMED ABADA
3. SEMESTER INFORMATIK
E-MAIL: MONSEFABADA@GMX.DE

Inhaltsverzeichnis

Semantik Beschreibung Methoden

Es ist üblich, zwischen der Syntax und der Semantik einer Programmier Sprache zu unterscheiden.

- Syntax beschäftigt sich mit der grammatikalischen Struktur von Programmen.
- Semantik beschäftigt sich mit der Bedeutung eines (grammatikalisch korrekten) Programms.

z : = x ; x : = y ; y: = z

Die Syntaktische Analyse des Programms :
Das Programm besteht aus drei Anweisungen, die von einander durch '' ; '' getrennt sind. Jede dieser Anweisungen hat die Form von einer Variabel, gefolgt von Dem Symbol '':= '' und einer weitern Variable.
Semantik:
Die Bedeutung des gezeigten Programms ist, die Werte der Variablen x und y auszutauschen.

Operational Semantics

Individuelle Anweisungen, getrennt durch '' ; '' sind nacheinander, von links nach rechts auszuführen.

Um eine Anweisung der Form '' a : = b '' auszuführen, ermitteln wir den Wert der zweiten Variablen '' b'' und ordnen ihren Wert der Variablen a (der ersten Variablen) zu.

Beispiel:

$$< z := x \ ; \ x := y \ ; \ y := z \ , \ [x \to 5, y \to 7, z \to 0] >$$

$$\Rightarrow \quad < x := y \ ; \ y := z \ , \ [x \to 5, y \to 7, z \to 5] >$$

$$\Rightarrow \quad < y := z \ , \ [x \to 7, y \to 7, z \to 5] >$$

$$\Rightarrow \quad < [x \to 7, y \to 5, z \to 5] >$$

$< z := x , S_0 > \rightarrow S_1$ $< x := y , S_1 > \rightarrow S_2$

$< z := x ; x := y, S_0 > \rightarrow S_2$ $< y := z, S_2 > \rightarrow S_3$

$< z := x ; x := y; y := z, S_0 > \rightarrow S_3$

Abkürzungen:

$S_0 = [x \rightarrow 5, y \rightarrow 7, z \rightarrow 0]$

$S_1 = [x \rightarrow 5, y \rightarrow 7, z \rightarrow 5]$

$S_2 = [x \rightarrow 7, y \rightarrow 7, z \rightarrow 5]$

$S_3 = [x \rightarrow 7, y \rightarrow 5, z \rightarrow 5]$

$< z := x ; x := y; y := z, S_0 > \rightarrow S_3$

Denotational Semantics

Die Bedeutungen werden durch mathematische Objekte modelliert, die den Effekt der Ausführung dieses Konstrukts (z.B. Prog.) wiedergeben.

Nur der Effekt ist von Interesse, nicht der Weg dorthin.

Für unser Beispielprogramm erhalten wir die Funktionen:

Der Effekt einer Sequenz von ausdrücken, die durch ’’; ’’ voneinander getrennt sind, ist Zusammengesetzt aus dem Effekten der einzelnen Effekte(’’ functional Composition ’’)

Der Effekt eines Ausdrucks der Form ’’ a = b ’’ ist so, dass ein Ausgangszustand einen Zustand produziert: der neue Zustand unterscheidet sich von dem Alten dadurch, dass der Wert der Ersten Variablen (a) gleich dem Wert der Zweiten variablen (b) ist.

$\mathcal{S}$[z:=x], $\mathcal{S}$[x:=y], und $\mathcal{S}$[y:=z] für jeden Ausdruck.

Und für das gesamte Programm :

$$\mathcal{S}[\![\mathrm{z:=x;\ x:=y;\ y:=z}]\!] = \mathcal{S}[\![\mathrm{y:=z}]\!] \circ \mathcal{S}[\![\mathrm{x:=y}]\!] \circ \mathcal{S}[\![\mathrm{z:=x}]\!]$$

Beispiel:

$$\begin{aligned}
&\mathcal{S}[\![\mathrm{z:=x;\ x:=y;\ y:=z}]\!]([\mathrm{x}\mapsto\mathbf{5},\ \mathrm{y}\mapsto\mathbf{7},\ \mathrm{z}\mapsto\mathbf{0}]) \\
&\quad = (\mathcal{S}[\![\mathrm{y:=z}]\!] \circ \mathcal{S}[\![\mathrm{x:=y}]\!] \circ \mathcal{S}[\![\mathrm{z:=x}]\!])([\mathrm{x}\mapsto\mathbf{5},\ \mathrm{y}\mapsto\mathbf{7},\ \mathrm{z}\mapsto\mathbf{0}]) \\
&\quad = \mathcal{S}[\![\mathrm{y:=z}]\!](\mathcal{S}[\![\mathrm{x:=y}]\!](\mathcal{S}[\![\mathrm{z:=x}]\!]([\mathrm{x}\mapsto\mathbf{5},\ \mathrm{y}\mapsto\mathbf{7},\ \mathrm{z}\mapsto\mathbf{0}]))) \\
&\quad = \mathcal{S}[\![\mathrm{y:=z}]\!](\mathcal{S}[\![\mathrm{x:=y}]\!]([\mathrm{x}\mapsto\mathbf{5},\ \mathrm{y}\mapsto\mathbf{7},\ \mathrm{z}\mapsto\mathbf{5}])) \\
&\quad = \mathcal{S}[\![\mathrm{y:=z}]\!]([\mathrm{x}\mapsto\mathbf{7},\ \mathrm{y}\mapsto\mathbf{7},\ \mathrm{z}\mapsto\mathbf{5}]) \\
&\quad = [\mathrm{x}\mapsto\mathbf{7},\ \mathrm{y}\mapsto\mathbf{5},\ \mathrm{z}\mapsto\mathbf{5}]
\end{aligned}$$

Axiomatic Semantics

Ein Programm ist abschnittsweise korrekt (pratially correct) bezüglich einer gegebenen Vorbedingung und einer „ Nachbedingung “, wenn der Ausgangs-Zustand die Vorbedingung erfüllt und das Programm beendet wird und der Endzustand die Nachbedingung erfüllt.

- Für unser Beispielprogramm haben wir die partial correctness property
  $\{x = n \wedge y = m\}\ z := x\ ;\ x := y\ ;\ y := z\ \{y = n \wedge x = m\}$
  Vorbedingung Nachbedingung

  Der Zustand $[x \mapsto 5, y \mapsto 7, z \mapsto 0]$ erfüllt die Vorbedingung indem er $n = 5, m = 7$ setzt.wenn wir die partial correctness property bewiesen haben, können wir daraus schließen, das, falls das Programm beendet wird, dann ein Zustand herauskommst, bei dem y = 5 und x_= 7 ist.

- Axiomatische Semantik bietet ein logisches System, um Partial Correctness für einzelne Programme zu beweisen.

Beweisbaum für obiges Beispiel :

$$\frac{\dfrac{\{P_0\}\ z := x\ \{P_1\} \qquad \{P_1\}\ x := y\ \{P_2\}}{\{P_0\}\ z := x\ ; x := y\ \{P_2\}} \qquad \{P_2\}\ y := z\ \{P_2\}}{\{P_0\}\ z := x\ ;\ x := y\ ;\ y := z\ \{P_3\}}$$

Hier bei werden folgende Abkürzungen verwendet :

$P_0 = x = n \wedge y = m$

$P_1 = z = n \wedge y = m$

$P_2 = z = n \wedge x = m$

$P_3 = y = n \wedge x = m$

- Wir werden für jeden der Ansätze (Operational, denotational, axiomatic Semantics) eine einfache (Programm) Sprache von While - Programmen entwickeln.

Die Beispiel – Sprache WHILE

Für unsere Sprache gibt es folgende Meta-Variablen und Kategorien:

- **numerals**

  n ∈ **Num**

- **variables**

  x ∈ **VAR**

- **arithmetic expressions**

  a ∈ **Aexp** (Arithmetische Ausdrücke)

  $a ::= n \mid x \mid a_1 + a_2 \mid a_1 * a_2 \mid a_1 - a_2$

- **booleans expressins**

  b ∈ **Bexp** (bool'sche Ausdrücke)

  $b ::= \text{True} \mid \text{False} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2$

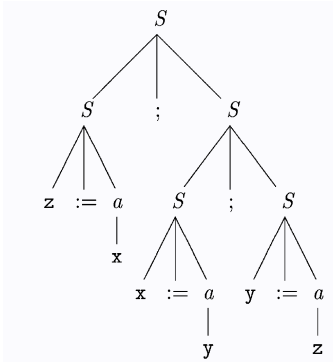
- **statements**

  S ∈ **Stm**
  $S ::= x := a \mid \text{Skip} \mid S_1 ; S_2 \mid \text{if b then } S_1 \text{ else } S_2 \mid \text{While b do S}$

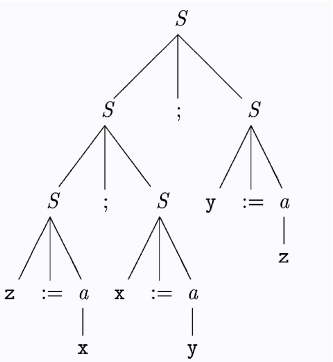
Die Meta-Variablen können auch mit Zusätzen versehen werden : Beispielsweise beziehen sich n, n', n_1, n_2 allesamt auf Ziffern.

Z := x ; x := Y ; Y := z

z : = x; (x : = y ; y : = z)



(z : = x ; x : = y); y : = z



Für Statments schreibt man oft die Klammern als begin....end

Die Semantik von Ausdrücken

n ::= 0 | 1 | n 0 | n 1

um die Zahl zu ermitteln, die durch eine Ziffer repräsentiert wird, wollen wir die folgende Funktion definieren:

***N* : Num ⟶ Z**

Diese wird als *Semantik Funktion* bezeichnet, weil sie die Semantik von Ziffern definiert .Wir wollen, dass *N* eine vollständige Funktion ist, weil wir eine eindeutige Zahl für jede Ziffer von Num ermitteln wollen**.**

Falls n ∈ *N* ist, schreiben wir *N* [[n]] um *N* auf n.

Die semantische Funktion *N* wird durch die folgenden semantischen Sätze oder Gleichungen definiert :

N [[0]] = 0

N [[1]] = 1

N [[n 0]] = 2 * *N* [[n]]

N [[n 1]] = 2 * *N* [[n]] + 1

Beispiel :

N [[101]] = 2 * *N* [[10]] + 1
= 2 * 2 * *N*[[1]] + 1
= 5

Semantische Funktionen

Die Bedeutung eines Ausdrucks hängt von den Werten ab, mit denen Variablen belegt sind, die in dem Ausdruck vorkommen. Wenn beispielsweise x mit dem Wert 3 belegt ist, ergibt der arithmetische Ausdruck x + 1 den Wert 4.Wenn aber x mit dem Wert 2 belegt ist, ergibt die gleiche Funktion den wert 3.Wir führen deshalb das Konzept der Zustände ein : zu jeder Variablen nimmt der Zustand einen jeweils gültigen Wert an.

Beispiel:

- **Zustand = var → z**

X	5
Y	7
z	0

Oder, [x → 5, y → 7, z → 0]

- ***A* : Aexp → (Zustand → z)**

A[[n]] s = *N* [[n]]
A[[x]] s = s x
A[[a_1 + a_2]] s = *A*[[a_1]] s + *A*[[a_2]] s
A[[a_1 * a_2]] s = *A*[[a_1]] s * *A* [[a_2]] s
A[[a_1 - a_2]] s = *A*[[a_1]] s - *A* [[a_2]] s

Beispiel :

s x = 3

A[[x + 1]] s = *A*[[x]] s + *A*[[1]] s
= (s x) + *N* [[1]]
= 3 + 1
= 4

A[[-a]] s = ?

$$
\begin{aligned}
\mathcal{B}[\![\texttt{true}]\!]s &= \mathbf{tt} \\
\mathcal{B}[\![\texttt{false}]\!]s &= \mathbf{ff} \\
\mathcal{B}[\![a_1 = a_2]\!]s &= \begin{cases} \mathbf{tt} & \text{if } \mathcal{A}[\![a_1]\!]s = \mathcal{A}[\![a_2]\!]s \\ \mathbf{ff} & \text{if } \mathcal{A}[\![a_1]\!]s \neq \mathcal{A}[\![a_2]\!]s \end{cases} \\
\mathcal{B}[\![a_1 \leq a_2]\!]s &= \begin{cases} \mathbf{tt} & \text{if } \mathcal{A}[\![a_1]\!]s \leq \mathcal{A}[\![a_2]\!]s \\ \mathbf{ff} & \text{if } \mathcal{A}[\![a_1]\!]s > \mathcal{A}[\![a_2]\!]s \end{cases} \\
\mathcal{B}[\![\neg\ b]\!]s &= \begin{cases} \mathbf{tt} & \text{if } \mathcal{B}[\![b]\!]s = \mathbf{ff} \\ \mathbf{ff} & \text{if } \mathcal{B}[\![b]\!]s = \mathbf{tt} \end{cases} \\
\mathcal{B}[\![b_1 \wedge b_2]\!]s &= \begin{cases} \mathbf{tt} & \text{if } \mathcal{B}[\![b_1]\!]s = \mathbf{tt} \text{ and } \mathcal{B}[\![b_2]\!]s = \mathbf{tt} \\ \mathbf{ff} & \text{if } \mathcal{B}[\![b_1]\!]s = \mathbf{ff} \text{ or } \mathcal{B}[\![b_2]\!]s = \mathbf{ff} \end{cases}
\end{aligned}
$$

Eigenschaften der Semantik

Freie Variablen

Die freie Variable eines arithmetischen Ausdrucks a ist definiert als der Satz von Variablen, die darin vorkommen. Formell können wir eine compositinal Definition für diese Unter-Set FV (a) aus Var bestimmen.

FV (a) aus Var.

FV (n)	=	$\varnothing$
FV (x)	=	{ x }
FV ($a_1 + a_2$)	=	FV (a_1) U FV (a_2)
FV ($a_1 * a_2$)	=	FV (a_1) U FV (a_2)
FV ($a_1 - a_2$)	=	FV (a_1) U FV (a_2)

- Betrachten wir das Beispiel FV (x + 1) = { x } und FV (x + y + x) = { x, y }

Substitutionen (Ersetzungen)

Die formale Definition sieht folgendermaßen aus :

$$
\begin{array}{lcl}
n[y \mapsto a_0] & = & n \\
x[y \mapsto a_0] & = & \begin{cases} a_0 & \text{if } x = y \\ x & \text{if } x \neq y \end{cases} \\
(a_1 + a_2)[y \mapsto a_0] & = & (a_1[y \mapsto a_0]) + (a_2[y \mapsto a_0]) \\
(a_1 \star a_2)[y \mapsto a_0] & = & (a_1[y \mapsto a_0]) \star (a_2[y \mapsto a_0]) \\
(a_1 - a_2)[y \mapsto a_0] & = & (a_1[y \mapsto a_0]) - (a_2[y \mapsto a_0])
\end{array}
$$

Beispiel :

(x+1)[y→3] = 3 + 1

(x+y * x)[x→y -5] =(y - 5) + y * (y – 5)

$$
(s[\, y \rightarrow v\,])\, x = \begin{cases} v & \text{if } x = y \\ sx & \text{if } x \neq y \end{cases}
$$

Natürliche und strukturelle operationelle Semantik

Operational Semantik beschäftigt sich damit, wie ein Programm auszuführen ist, und nicht nur mit dem Ergebnis dieser Ausführung. Dazu gibt es zwei verschiedene Ansätze:

- Natürliche Semantik: beschreibt, wie das Endergebnis / Gesamtergebnis von Ausführungen erhalten wird.
- Strukturelle operationelle Semantik : beschreibt die einzelnen Schritte der Berechnungen.

''While'' eignet sich für beide Arten von Semantik.

Natürliche Semantik

Natürliche Semantik beschäftigt sich mit der Beziehung zwischen Ein- und Ausgangszustand einer Programmausführung.

[ass $_{ns}$] $< x := a, s > \rightarrow s [x \mapsto A[[a]] s]$

[Skip $_{ns}$] $< Skip , s > \rightarrow s$

[Comp $_{ns}$] $$\frac{< S_1 , s > \rightarrow s' , < S_2 , s' > \rightarrow s''}{< S_1 ; S_2 , s > \rightarrow s''}$$

[if $^{tt}_{ns}$] $$\frac{< S_1 , s > \rightarrow s'}{< \text{if b then } S_1 \text{ else } S_2 , s > \rightarrow s'} \quad \text{if } B[[b]] s = tt$$

[if $^{ff}_{ns}$] $$\frac{< S_2 , s > \rightarrow s'}{< \text{if b then } S_1 \text{ else } S_2 , s > \rightarrow s'} \quad \text{if } B[[b]] = ff$$

[while $^{tt}_{ns}$] $$\frac{< S , s > \rightarrow s' , < \text{while b do S} , s' > \rightarrow S''}{< \text{while b do S} , s > \rightarrow s''} \quad \text{if } B[[b]] s = tt$$

[while $^{ff}_{ns}$] $< \text{while b do S}, s' > \rightarrow s$ if $B[[b]] = ff$

Eigenschaften der Semantik

'' while b do S '' ist semantic equivalent zu '' if b then (S ; while b do S) else skip ''
Semantik äquivalent? das heißt, dass für alle Zustände s und s' gilt:
$< S_1, s > \rightarrow s'$ a $< {}_2, s > \rightarrow s'$

- < while b do S, s >→ S'' (*)
- < if b then (S ; while b do S) else skip, s >→ S'' (**)

*weil (*) gilt → einen derivation tree T haben→[while $^{f}_{ns}$] oder [while $^{tt}_{ns}$]*

weil (*) gilt, wissen wir, dass wir einen derivation tree T dafür haben. Er kann zwei Formen haben, je nachdem ob er konstruiert wurde mit *[while $^{ff}_{ns}$] oder [while $^{tt}_{ns}$]*

T1 T2

, wobei T_1 eine dervation tree mit der Wurzel< S, s >→s'
T_2 eine dervation Tree mit der Wurzel

< while b do S , s >→ S'' **< while b do S , s' >→ S''**

T_1 und T_2 für die Regeln [$comp_{ns}$] voraussetzen :

T1 T2

<S ; while b do S , s >→ S''

wenn B[[b]]s = tt → [if $^{tt}_{ns}$]

T1 T2

<S ; while b do S , s >→ S''

< if b then (S ; while b do S , s) else skip, s > → S''

Wenn man weiß, dass B[[b]] s = tt. Können wir die Regel [if $^{tt}_{ns}$] benutzen um den *derivation tree* zu konstruieren.

'' Das beweist, das (**) gilt .''
Alternativ ist der derivation tree T ein konkreter Fall von [While $^{ff}_{ns}$]
Wenn B[[b]]s = ff , und es muss gelten, dass s = s' ist.
Dann ist T:
< while b do S, s > → s [$skip_{ns}$] dann < skip , s > → s'' [if $^{ff}_{ns}$] dann < skip, s >→ s''

(* *) gilt → derivation tree T→[if $^{tt}_{ns}$] oder [if $^{ff}_{ns}$]
wenn B [[b]] s = tt → T_1 mit der Wurzel < S ; while b do S , s >→ s''
< S_1 ; S_2 , s >→ s''

[$comp_{ns}$] → (T_2 < S, s > → s' und T_3 < while b do S, s'> → s'')
jetzt kann man einfach [while $^{tt}_{ns}$] benützen, um T_2 und T_3 zu einem der derivation Tree für (*) zu kombinieren .

Im zweiten Fall: B[[b]] s = ff , und T wird mit [if $^{ff}_{ns}$] konstruiert.
< skip , s >→ s'' mit dem Axiom [$skip_{ns}$] müss gelten, dass s = s'ist → [while ^{ff}ns] → (*)

Strukturelle operationelle Semantik

[ass $_{sos}$] $\quad < x := a, s > \rightarrow s\,[\,x \rightarrow \mathcal{A}[[a]]\,s\,]$

[Skip $_{sos}$] $\quad < Skip\ , s > \rightarrow s$

[comp ^{1} $_{sos}$]
$$\frac{< S_1\ , s > \Rightarrow < S'_1\ , s' >}{< S_1\ ; S_2\ , s > \Rightarrow < S'_1\ ; S_2\ , s' >}$$

[comp ^{2} $_{sos}$]
$$\frac{< S_1\ , s > \Rightarrow s'}{< S_1\ ; S_2\ , s > \Rightarrow < S_2\ , s' >}$$

[if $^{tt}_{sos}$] $\quad < \text{if b then } S_1 \text{ else } S_2\ , s > \Rightarrow < S_1\ , s >$ if $\mathcal{B}\,[[\,b\,]]\,s = tt$

[if $^{ff}_{sos}$] $\quad < \text{if b then } S_1 \text{ else } S_2\ , s > \Rightarrow < S_2\ , s >$ if $\mathcal{B}\,[[\,b\,]]\,s = ff$

[while $_{sos}$] $\quad < \text{while b do S}\ , s > \Rightarrow$

$< \text{if b then (S ; while b do S, s') else skip, s} >$

Beispiel 1:

(z := x; x := y); y := z

Is$_0$ x = 5 und s$_0$ y=7 . Dann erhalten wir die Herleitungsfolge:

$$\begin{aligned}&\langle (z := x;\ x := y);\ y := z,\ s_0\rangle\\ &\quad\Rightarrow \langle x := y;\ y := z,\ s_0[z\mapsto 5]\rangle\\ &\quad\Rightarrow \langle y := z,\ (s_0[z\mapsto 5])[x\mapsto 7]\rangle\\ &\quad\Rightarrow ((s_0[z\mapsto 5])[x\mapsto 7])[y\mapsto 5]\end{aligned}$$

$$\langle (z := x;\ x := y);\ y := z,\ s_0\rangle \Rightarrow \langle x := y;\ y := z,\ s_0[z\mapsto 5]\rangle$$

Der Herleitungsbaum ist konstruiert aus dem Grundsatz [ass $_{sos}$] und den Regeln [comp $^{1}_{sos}$] und [Comp $^{2}_{sos}$].

$$\dfrac{\dfrac{\langle z := x,\ s_0\rangle \Rightarrow s_0[z\mapsto 5]}{\langle z := x;\ x := y,\ s_0\rangle \Rightarrow \langle x := y,\ s_0[z\mapsto 5]\rangle}}{\langle (z := x;\ x := y);\ y := z,\ s_0\rangle \Rightarrow \langle x := y;\ y := z,\ s_0[z\mapsto 5]\rangle}$$

Der Herleitungsbaum für den zweiten Schritt ist ähnlich konstruiert und benutzt nur [ass $_{sos}$] und [comp $^{2}_{sos}$] und der dritte Schritt ist einfach eine Instanz von [ass $_{sos}$].

Beispiel 2:

Nehmen wir an, dass s x = 3 ist.

$$\langle \mathtt{y{:=}1;\ while}\ \neg(\mathtt{x{=}1})\ \mathtt{do}\ (\mathtt{y{:=}y}\star\mathtt{x;\ x{:=}x{-}1}),\ s\rangle$$

$$\langle \mathtt{while}\ \neg(\mathtt{x{=}1})\ \mathtt{do}\ (\mathtt{y{:=}y}\star\mathtt{x;\ x{:=}x{-}1}),\ s[\mathtt{y}\mapsto\mathbf{1}]\rangle$$

dieses wird mit dem Grundsatz [ass $_{sos}$] und der Regel [comp$^2_{sos}$] erzielt und im Herleitungsbaum aufgezeigt:

$$\frac{\langle \mathtt{y{:=}1},\ s\rangle \Rightarrow s[\mathtt{y}\mapsto\mathbf{1}]}{\langle \mathtt{y{:=}1;\ while}\ \neg(\mathtt{x{=}1})\ \mathtt{do}\ (\mathtt{y{:=}y}\star\mathtt{x;\ x{:=}x{-}1}),\ s\rangle \Rightarrow \langle \mathtt{while}\ \neg(\mathtt{x{=}1})\ \mathtt{do}\ (\mathtt{y{:=}y}\star\mathtt{x;\ x{:=}x{-}1}),\ s[\mathtt{y}\mapsto\mathbf{1}]\rangle}$$

Der nächste Schritt der Ausführung wird die Schleife neu schreiben mit der bedingten Benutzung des Grundsatzes [while $_{sos}$] mit der wir die Konfiguration bekommen

$$\langle \mathtt{if}\ \neg(\mathtt{x{=}1})\ \mathtt{then}\ ((\mathtt{y{:=}y}\star\mathtt{x;\ x{:=}x{-}1});\ \mathtt{while}\ \neg(\mathtt{x{=}1})\ \mathtt{do}\ (\mathtt{y{:=}y}\star\mathtt{x;\ x{:=}x{-}1}))\ \mathtt{else\ skip},\ s[\mathtt{y}\mapsto\mathbf{1}]\rangle$$

Der folgende Schritt wird den Test ausführen und ergibt (gemäß [if $^{tt}_{sos}$]) die Konfiguration

$$\langle (\mathtt{y{:=}y}\star\mathtt{x;\ x{:=}x{-}1});\ \mathtt{while}\ \neg(\mathtt{x{=}1})\ \mathtt{do}\ (\mathtt{y{:=}y}\star\mathtt{x;\ x{:=}x{-}1}),\ s[\mathtt{y}\mapsto\mathbf{1}]\rangle$$

Wir können dann [ass $_{sos}$], [comp $^2_{sos}$] benutzen und [comp $^1_{sos}$] bekommen die Konfiguration

$$\langle \mathtt{x{:=}x{-}1;\ while}\ \neg(\mathtt{x{=}1})\ \mathtt{do}\ (\mathtt{y{:=}y}\star\mathtt{x;\ x{:=}x{-}1}),\ s[\mathtt{y}\mapsto\mathbf{3}]\rangle$$

wie es durch den Herleitungsbaum bestätigt wird:

$$\frac{\langle \mathtt{y{:=}y}\star\mathtt{x},\ s[\mathtt{y}\mapsto\mathbf{1}]\rangle \Rightarrow s[\mathtt{y}\mapsto\mathbf{3}]}{\langle \mathtt{y{:=}y}\star\mathtt{x;\ x{:=}x{-}1},\ s[\mathtt{y}\mapsto\mathbf{1}]\rangle \Rightarrow \langle \mathtt{x{:=}x{-}1},\ s[\mathtt{y}\mapsto\mathbf{3}]\rangle}$$

$$\frac{}{\langle (\mathtt{y{:=}y}\star\mathtt{x;\ x{:=}x{-}1});\ \mathtt{while}\ \neg(\mathtt{x{=}1})\ \mathtt{do}\ (\mathtt{y{:=}y}\star\mathtt{x;\ x{:=}x{-}1}),\ s[\mathtt{y}\mapsto\mathbf{1}]\rangle \Rightarrow \langle \mathtt{x{:=}x{-}1;\ while}\ \neg(\mathtt{x{=}1})\ \mathtt{do}\ (\mathtt{y{:=}y}\star\mathtt{x;\ x{:=}x{-}1}),\ s[\mathtt{y}\mapsto\mathbf{3}]\rangle}$$

mit [ass $_{sos}$] und [comp $^2_{sos}$] erhalten wir die nächste Konfiguration

$$\langle \mathtt{while}\ \neg(\mathtt{x{=}1})\ \mathtt{do}\ (\mathtt{y{:=}y}\star\mathtt{x;\ x{:=}x{-}1}),\ s[\mathtt{y}\mapsto\mathbf{3}][\mathtt{x}\mapsto\mathbf{2}]\rangle$$

nachfolgend erhalten die Konfiguration den abschließenden Status $s[\mathtt{y}\mapsto\mathbf{6}][\mathtt{x}\mapsto\mathbf{1}]$.