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 13 (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:

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

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:

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

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 $\langle \_,\ \_\rangle \to_1 \langle \_,\ \_\rangle$ muss ebenfalls um Regeln für c_1 `or` c_2 ergänzt werden:

$$\textsc{Or1}_{\textsc{ss}}\colon \langle c_1 \texttt{ or } c_2,\ \sigma\rangle \to_1 \langle c_1,\ \sigma\rangle \qquad \textsc{Or2}_{\textsc{ss}}\colon \langle c_1 \texttt{ or } c_2,\ \sigma\rangle \to_1 \langle c_2,\ \sigma\rangle$$

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 nichtdetermistischen 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 Ausnahmen While_X

Die Sprache While soll um Ausnahmen und deren Behandlung erweitert werden. Dazu werden zwei neue Anweisungen zu While hinzugefügt:

$$\texttt{raise}\ X \qquad \text{und} \qquad \texttt{try}\ c_1\ \texttt{catch}\ X\ c_2$$

Dabei bezeichne X den Namen der ausgelösten bzw. behandelten Ausnahme und Xcp die Menge aller Namen von Ausnahmen. Wie schon bei Variablennamen ist die konkrete Notation für diese Namen irrelevant, im Folgenden werden wieder alphanumerische Zeichenfolgen verwendet.

Wenn in c_1 die Ausnahme X mittels `raise` X erzeugt wird, soll der Rest von c_1 nicht mehr abgearbeitet, sondern der Exception-Handler c_2 der Anweisung `try` c_1 `catch` X c_2 ausgeführt werden. Wird die Exception X nicht ausgelöst, wird c_2 ignoriert.[2]

5.4.1 Small-Step-Semantik

Neue Regeln der Small-Step-Semantik:

$$\textsc{Try}_{SS}\colon \frac{\langle c_1,\ \sigma\rangle \to_1 \langle c_1',\ \sigma'\rangle}{\langle \texttt{try}\ c_1\ \texttt{catch}\ X\ c_2,\ \sigma\rangle \to_1 \langle \texttt{try}\ c_1'\ \texttt{catch}\ X\ c_2,\ \sigma'\rangle}$$

$$\textsc{TryCatch}\colon \langle \texttt{try raise}\ X\ \texttt{catch}\ X\ c,\ \sigma\rangle \to_1 \langle c,\ \sigma\rangle$$

[2] Dieser Mechanismus ähnelt den Exceptions von Java, es gibt allerdings keine Subsumtionsbeziehung zwischen verschiedenen Ausnahmenamen wie in folgendem Beispiel: `try { throw new FileNotFoundException(); } catch (IOException _) { ... }`. Außerdem sind Exceptions in Java Werte, die mit allen anderen Sprachmitteln kombiniert werden können, z. B. `Exception e = (... ? new ArrayStoreException() : new NullPointerException()); throw e;`. In While_X gibt es jede Exception nur einmal und der konkrete Name muss an der Auslösestelle selbst stehen.

$$\textsc{TryRaise}: \frac{X \neq X'}{\langle \texttt{try raise } X \texttt{ catch } X' \ c,\ \sigma\rangle \to_1 \langle \texttt{raise } X,\ \sigma\rangle}$$

$$\textsc{TrySkip}: \langle \texttt{try skip catch } X \ c,\ \sigma\rangle \to_1 \langle \texttt{skip},\ \sigma\rangle$$

$$\textsc{SeqRaise}: \langle \texttt{raise } X \texttt{ ; } \ c,\ \sigma\rangle \to_1 \langle \texttt{raise } X,\ \sigma\rangle$$

Man braucht Exception-Propagationsregeln wie SeqRaise und TryRaise für alle zusammengesetzten Ausdrücke, für die man auch Teilausdruckreduktionsregeln hat. Im Falle von While_X ist dies nur die Sequenz.

Alle bisherigen Regeln gelten weiterhin und brauchen nicht angepasst zu werden. Damit ist diese Erweiterung modular.

Lemma 20 (Fortschrittslemma).
Wenn $c \neq \texttt{skip}$ und $\forall X.\ c \neq \texttt{raise } X$, dann gibt es c' und σ' mit $\langle c,\ \sigma\rangle \to_1 \langle c',\ \sigma'\rangle$.

Beweis. Beweis durch Induktion über c:

- Fälle `skip`, `raise` X: Explizit ausgeschlossen.
- Fälle x `:=` a, `if (`b`) then` c_1 `else` c_2, `while (`b`) do` c: Gleich wie im Fortschrittslemma 3 für While.
- Fall c_1`;` c_2: Induktionsannahme: Wenn $c_1 \neq \texttt{skip}$ und $\forall X.\ c_1 \neq \texttt{raise } X$, dann gibt es ein c_1' und σ' mit $\langle c_1,\ \sigma\rangle \to_1 \langle c_1',\ \sigma'\rangle$.

  Zu zeigen: Es gibt ein c' und σ' mit $\langle c_1\texttt{;}\ c_2,\ \sigma\rangle \to_1 \langle c',\ \sigma'\rangle$. Fallunterscheidung nach $c_1 = \texttt{skip}$ oder $c_1 = \texttt{raise } X$.
  - Fall $c_1 = \texttt{skip}$: (analog zum Beweis in Lem. 3)
    Nach Regel $\textsc{Seq2}_{\text{SS}}$ gilt $\langle \texttt{skip;}\ c_2,\ \sigma\rangle \to_1 \langle c_2,\ \sigma\rangle$. Wähle also $c' = c_2$ und $\sigma' = \sigma$.
  - Fall $c_1 = \texttt{raise } X$: Nach Regel SeqRaise gilt $\langle \texttt{raise } X\texttt{;}\ c_2,\ \sigma\rangle \to_1 \langle \texttt{raise } X,\ \sigma\rangle$. Wähle also $c' = \texttt{raise } X$ und $\sigma' = \sigma$.
  - Fall $c_1 \neq \texttt{skip}$ und $\forall X.\ c_1 \neq \texttt{raise } X$: Nach Induktionsannahme gibt es ein c_1' und σ' mit $\langle c_1,\ \sigma\rangle \to_1 \langle c_1',\ \sigma'\rangle$. Nach Regel $\textsc{Seq1}_{\text{SS}}$ folgt damit $\langle c_1\texttt{;}\ c_2,\ \sigma\rangle \to_1 \langle c_1'\texttt{;}\ c_2,\ \sigma'\rangle$.
- Fall `try` c_1 `catch` Y c_2: Induktionsannahme: Wenn $c_1 \neq \texttt{skip}$ und $\forall X.\ c_1 \neq \texttt{raise } X$, dann gibt es c_1' und σ' mit $\langle c_1,\ \sigma\rangle \to_1 \langle c_1',\ \sigma'\rangle$.

  Zu zeigen: Es gibt ein c' und σ' mit $\langle \texttt{try } c_1 \texttt{ catch } Y\ c_2,\ \sigma\rangle \to_1 \langle c',\ \sigma'\rangle$.
  Fallunterscheidung nach $c_1 = \texttt{skip}$ oder $c_1 = \texttt{raise } X$.
  - Fall $c_1 = \texttt{skip}$: Nach Regel TrySkip gilt $\langle \texttt{try skip catch } Y\ c_2,\ \sigma\rangle \to_1 \langle \texttt{skip},\ \sigma\rangle$. Wähle also $c' = \texttt{skip}$ und $\sigma' = \sigma$.
  - Fall $c_1 = \texttt{raise } X$:
    Wenn $X = Y$, dann gilt nach Regel TryCatch, dass $\langle \texttt{try raise } X \texttt{ catch } Y\ c_2,\ \sigma\rangle \to_1 \langle c_2,\ \sigma\rangle$; wähle also $c' = c_2$ und $\sigma' = \sigma$. Wenn $X \neq Y$, dann gilt nach Regel TryRaise, dass $\langle \texttt{try raise } X \texttt{ catch } Y\ c_2,\ \sigma\rangle \to_1 \langle \texttt{raise } X,\ \sigma\rangle$; wähle also $c' = \texttt{raise } X$ und $\sigma' = \sigma$.
  - Fall $c_1 \neq \texttt{skip}$ und $\forall X.\ c_1 \neq \texttt{raise } X$:
    Nach Induktionsannahme gibt es ein c_1' und σ' mit $\langle c_1,\ \sigma\rangle \to_1 \langle c_1',\ \sigma'\rangle$.
    Nach Regel $\textsc{Try}_{SS}$ folgt damit $\langle \texttt{try } c_1 \texttt{ catch } Y\ c_2,\ \sigma\rangle \to_1 \langle \texttt{try } c_1' \texttt{ catch } Y\ c_2,\ \sigma'\rangle$. □

5.4.2 Big-Step-Semantik

Um Exception-Handling als Big-Step-Semantik anzugeben, muss man in den Regeln die verschiedenen Terminationsarten einer Anweisung (`skip` und `raise` _) angeben können. Dazu muss man den Typ der

Auswertungsrelation $\langle \_,\ \_\rangle \Downarrow \_$ ändern auf

$$\langle \_,\ \_\rangle \Downarrow \_ \subseteq \mathsf{Com} \times (\mathsf{Xcp}' \times \Sigma) \times (\mathsf{Xcp}' \times \Sigma)$$

wobei Xcp' die Menge der Exceptionnamen Xcp um `None` erweitert. `None` bezeichne, dass gerade *keine* unbehandelte Exception vorliegt; ansonsten speichert das neue Exception-Flag im Zustand (Variablenkonvention ξ), welche Exception ausgelöst wurde.

Damit ändern sich die bisherigen Regeln wie folgt:

$$\textsc{Skip}_{BS}\colon \langle \texttt{skip},\ (\texttt{None}, \sigma)\rangle \Downarrow (\texttt{None}, \sigma) \qquad \textsc{Ass}_{BS}\colon \langle x\ \texttt{:=}\ a,\ (\texttt{None}, \sigma)\rangle \Downarrow (\texttt{None}, \sigma[x \mapsto \mathcal{A}[\![a]\!]\,\sigma])$$

$$\textsc{Seq}_{BS}\colon \frac{\langle c_0,\ (\texttt{None}, \sigma)\rangle \Downarrow (\xi', \sigma') \qquad \langle c_1,\ (\xi', \sigma')\rangle \Downarrow (\xi'', \sigma'')}{\langle c_0\,\texttt{;}\ c_1,\ (\texttt{None}, \sigma)\rangle \Downarrow (\xi'', \sigma'')}$$

$$\textsc{IfTT}_{BS}\colon \frac{\mathcal{B}[\![b]\!]\,\sigma = \mathbf{tt} \qquad \langle c_0,\ (\texttt{None}, \sigma)\rangle \Downarrow (\xi', \sigma')}{\langle \texttt{if (}b\texttt{) then}\ c_0\ \texttt{else}\ c_1,\ (\texttt{None}, \sigma)\rangle \Downarrow (\xi', \sigma')}$$

$$\textsc{IfFF}_{BS}\colon \frac{\mathcal{B}[\![b]\!]\,\sigma = \mathbf{ff} \qquad \langle c_1,\ (\texttt{None}, \sigma)\rangle \Downarrow (\xi', \sigma')}{\langle \texttt{if (}b\texttt{) then}\ c_0\ \texttt{else}\ c_1,\ (\texttt{None}, \sigma)\rangle \Downarrow (\xi', \sigma')}$$

$$\textsc{WhileFF}_{BS}\colon \frac{\mathcal{B}[\![b]\!]\,\sigma = \mathbf{ff}}{\langle \texttt{while (}b\texttt{) do}\ c,\ (\texttt{None}, \sigma)\rangle \Downarrow (\mathbf{ff}, \sigma)}$$

$$\textsc{WhileTT}_{BS}\colon \frac{\mathcal{B}[\![b]\!]\,\sigma = \mathbf{tt} \qquad \langle c,\ (\texttt{None}, \sigma)\rangle \Downarrow (\xi', \sigma') \qquad \langle \texttt{while (}b\texttt{) do}\ c,\ (\xi', \sigma')\rangle \Downarrow (\xi'', \sigma'')}{\langle \texttt{while (}b\texttt{) do}\ c,\ (\texttt{None}, \sigma)\rangle \Downarrow (\xi'', \sigma'')}$$

Neue Regeln:

$$\textsc{Raise}\colon \langle \texttt{raise}\ X,\ (\texttt{None}, \sigma)\rangle \Downarrow (X, \sigma) \qquad \textsc{Propagate}\colon \langle c,\ (X, \sigma)\rangle \Downarrow (X, \sigma)$$

$$\textsc{Try}_{BS}\colon \frac{\langle c_1,\ (\texttt{None}, \sigma)\rangle \Downarrow (\xi', \sigma') \qquad \xi' \neq X}{\langle \texttt{try}\ c_1\ \texttt{catch}\ X\ c_2,\ (\texttt{None}, \sigma)\rangle \Downarrow (\xi', \sigma')}$$

$$\textsc{Catch}\colon \frac{\langle c_1,\ (\texttt{None}, \sigma)\rangle \Downarrow (X, \sigma') \qquad \langle c_2,\ (\texttt{None}, \sigma)\rangle \Downarrow (\xi'', \sigma'')}{\langle \texttt{try}\ c_1\ \texttt{catch}\ X\ c_2,\ (\texttt{None}, \sigma)\rangle \Downarrow (\xi'', \sigma'')}$$

Einzig die Regel Propagate lässt sich bei gesetztem Exception-Flag anwenden. Alle anderen sind so formuliert, dass sie sich nur anwenden lassen, wenn das Exception-Flag ξ nicht gesetzt ist. Dadurch ist offensichtlich, dass die Big-Step-Semantik weiterhin deterministisch ist.

Bemerkung: Big-Step- und Small-Step-Semantik sind weiterhin in folgendem Sinne äquivalent:

1. Gelte $\langle c,\ (\texttt{None}, \sigma)\rangle \Downarrow (\xi', \sigma')$. Falls $\xi' = \texttt{None}$, dann $\langle c,\ \sigma\rangle \xrightarrow{*}_1 \langle \texttt{skip},\ \sigma'\rangle$. Falls $\xi' = X$, dann $\langle c,\ \sigma\rangle \xrightarrow{*}_1 \langle \texttt{raise}\ X,\ \sigma'\rangle$.
2. Wenn $\langle c,\ \sigma\rangle \xrightarrow{*}_1 \langle \texttt{skip},\ \sigma'\rangle$, dann $\langle c,\ (\texttt{None}, \sigma)\rangle \Downarrow (\texttt{None}, \sigma')$. Wenn $\langle c,\ \sigma\rangle \xrightarrow{*}_1 \langle \texttt{raise}\ X,\ \sigma'\rangle$, dann $\langle c,\ (\texttt{None}, \sigma)\rangle \Downarrow (X, \sigma')$.

Die Beweise verlaufen ähnlich wie für Thm. 8 und 9.