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 5: Compiler Besprechung: 27.05.2019

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

(a) [`ASSN x 5`, `JMPF` -1 `true`, `JMP 2`, `ASSN y (x + 7)`] ist ein ASM-Programm.

(b) $[\texttt{JMP 0}] \vdash \langle 0,\ \sigma\rangle \overset{\infty}{\to}$

(c) comp(`if (x <= 0) then x := 0 - x; y := 0 else skip`) =
[`JMPF 3 (x <= 0)`, `ASSN x (0 - x)`, `ASSN y 0`]

(d) [`JMP 3`, `ASSN y 0`, `JMPF` -3 $(y$ `== 1)`, `ASSN y 1`] ist abgeschlossen.

(e) Wenn $P +\!\!+ P' \vdash \langle 0,\ \sigma\rangle \overset{*}{\to} \langle |P| + |P'|,\ \sigma'\rangle$, dann gibt es ein σ^* mit $P \vdash \langle 0,\ \sigma\rangle \overset{*}{\to} \langle |P|,\ \sigma^*\rangle$ und $P' \vdash \langle 0,\ \sigma^*\rangle \overset{*}{\to} \langle |P'|,\ \sigma'\rangle$.

(f) Zu jedem While-Programm gibt es ein semantisch äquivalentes ASM-Programm.

(g) Zu jedem ASM-Programm gibt es ein semantisch äquivalentes While-Programm.

Lösung:

(1a) Richtig.

(1b) Richtig.

(1c) Falsch. Auch für `if`-Abfragen mit leerem else-Zweig erzeugt der Compiler einen Sprung nach dem then-Zweig. Richtig ist: [`JMPF 4 (x <= 0)`, `ASSN x (0 - x)`, `ASSN y 0`, `JMP 1`]

(1d) Falsch. $P_2 =$ `JMPF` -3 $(y$ `== 1`$)$, aber $0 \not\leq 2 + (-3)$. Abgeschlossenheit ist eine rein statische Aussage, die auch für nicht erreichbare Sprünge und unerfüllbare Bedingungen gelten muss.

(1e) Falsch, Beispiel: $P = [\texttt{JMP 2}]$, $P' = [I]$. Dann $[\texttt{JMP 2}, I] \vdash \langle 0,\ \sigma\rangle \to \langle 2,\ \sigma\rangle$, aber $[\texttt{JMP 2}] \vdash \langle 0,\ \sigma\rangle \overset{*}{\not\to} \langle 1,\ \sigma^*\rangle$ für alle σ^*.
Wenn P und P' abgeschlossen wären, würde dies aus dem Aufteilungslemma 36 folgen: Es gäbe ein σ^* mit $P \vdash \langle 0,\ \sigma\rangle \overset{*}{\to} \langle |P|,\ \sigma^*\rangle$ und (i) $P +\!\!+ P' \vdash \langle |P|,\ \sigma^*\rangle \overset{*}{\to} \langle |P| + |P'|,\ \sigma'\rangle$. Nochmalige Anwendung des Lemmas auf (i) ergäbe ein σ^{**} mit $P' \vdash \langle 0,\ \sigma^*\rangle \overset{*}{\to} \langle |P'|,\ \sigma^{**}\rangle$ und (ii) $P +\!\!+ P' \vdash \langle |P| + |P'|,\ \sigma^{**}\rangle \overset{*}{\to} \langle |P| + |P'|,\ \sigma'\rangle$. Aus (ii) folgte durch Regelinversion, dass $\sigma^{**} = \sigma'$ wäre.

(1f) Richtig. comp(c) ist nach den Simulationstheoremen 30 und 37 semantisch äquivalent zu c.

(1g) Falsch. Das Ganze scheitert daran, dass es keine unbenutzten Variabelnamen gibt.[1] Grundsätzlich darf also der "Simulationsoverhead" unseres konstruierten While-Programm keinen eigenen Zustand halten; Semantische Äquivalenz fordert nämlich, dass vom ursprünglichen Programm unangetastete Variablen weiterhin unbenutzt bleiben. Eine theoretische VM für

[1] Da unsere Variablenwerte nicht-größenbeschränkte natürliche Zahlen sind, könnte man auch versuchen, weitere Bits bspw. in den untersten Stellen einer Variablen zu verstecken. Hätten wir eine native Divisionsoperation zur Verfügung, würde das unser Problem auch (auf hässliche Weise) lösen! Leider müssen While-Programme Division algorithmisch durchführen, und da solche Algorithmen ihren eigenen Zustand benötigen, beißt sich hier die Katze in den Schwanz.

ASM-Programme in While müsste einen Programmzähler oder eine Grundblock-ID halten, für welche einfach kein Platz ist. Eine direkte Transformation eines unstrukturierten Kontrollflussgraphen in einen (semantisch äquivalenten) strukturierten CFG führt hingegen *notwendigerweise*[2] neue boolsche Variablen ein. (Das Kontrollflusskonstrukt, das uns für diese Äquivalenz fehlt, ist das multi-level `break`-Statement, wie es bspw. Java hat)

Abschließend kann man noch sagen, dass wir an dieser Stelle an eine natürliche Unzulänglichkeit unseres Semantikbegriffs stoßen: auch intuitiv irrelevante Teile des Programmverhaltens, wie rein temporär benutzte Variablen, werden zur Semantik des Programms gezählt. Echte Programmiersprachen haben differenziertere Konzepte von beobachtbarem Verhalten, oder gar ein formales Speichermodell, so dass solches "semantisches Rauschen" erst gar nicht auftritt.

2. `repeat` c `until` b-Schleife (H)

Neben `while`-Schleifen sind auch `repeat`-Schleifen in vielen Programmiersprachen verbreitet. In dieser Aufgabe sei Com um die Produktion `repeat` c `until` b erweitert. Die Small-Step-Semantik für `repeat` c `until` b sei durch folgende Regel gegeben:

$$\textsc{Repeat}: \langle \texttt{repeat}\ c\ \texttt{until}\ b,\ \sigma\rangle \to_1 \langle c;\ \texttt{if}\ (b)\ \texttt{then skip else repeat}\ c\ \texttt{until}\ b,\ \sigma\rangle$$

(a) Geben Sie Regeln in der Big-Step-Semantik an, die `repeat` c `until` b das gleiche Verhalten geben wie Repeat in der Small-Step-Semantik. Welche Schritte wären für einen Äquivalenzbeweis nötig?

(b) Erweitern Sie die Definition des Compilers comp auf Repeat-Schleifen.

(c) Erweitern Sie den Korrektheitsbeweis des Compilers (Thm. 30 und 37) um Repeat-Schleifen.

Lösung:

(2a) Big-Step-Regeln (analog zur `while`-Schleife):

$$\textsc{RepeatTT}: \frac{\langle c,\ \sigma\rangle \Downarrow \sigma' \qquad \mathcal{B}[\![b]\!]\,\sigma' = \mathbf{tt}}{\langle \texttt{repeat}\ c\ \texttt{until}\ b,\ \sigma\rangle \Downarrow \sigma'}$$

$$\textsc{RepeatFF}: \frac{\langle c,\ \sigma\rangle \Downarrow \sigma' \qquad \mathcal{B}[\![b]\!]\,\sigma' = \mathbf{ff} \qquad \langle \texttt{repeat}\ c\ \texttt{until}\ b,\ \sigma'\rangle \Downarrow \sigma''}{\langle \texttt{repeat}\ c\ \texttt{until}\ b,\ \sigma\rangle \Downarrow \sigma''}$$

Schritte für einen Äquivalenzbeweis (analog zu den Fällen für `while`-Schleifen):

- Schleifenabwicklungslemma für `repeat` (vgl. Lem. 9):
  `repeat` c `until` b und c; `if` (b) `then skip else repeat` c `until` b sind in der Big-Step-Semantik äquivalent.
  Beweis durch Regelinversion (Transformation von Ableitungsbäumen).
- `repeat`-Fall im Induktionsbeweis der Simulation Big-Step ⇒ Small-Step (Thm. 21).
  Beweis analog zur `while`-Schleife (Übungsblatt 3)
- `repeat`-Fall im Induktionsbeweis der Simulation Small-Step ⇒ Big-step (Thm. 22):
  Beweis analog zur `while`-Schleife, braucht obiges Schleifenabwicklungslemma für `repeat`.

(2b) $\text{comp}(\texttt{repeat}\ c\ \texttt{until}\ b) = \text{comp}(c) +\!+ [\texttt{JMPF}\ -|\text{comp}(c)|\ b]$

(2c) Weder Verschiebungslemma (Lem. 32) noch Aufteilungslemma (Lem. 36) müssen neu bewiesen werden, da diese unabhängig von While und dem Compiler sind. Für das Abgeschlossenheitslemma der generierten Instruktionsliste (Lem. 35) muss noch der neue Fall für `repeat` gezeigt werden.

[2] siehe Ashcroft und Manna, "The translation of "go to" programs to "while" programs", 1971

Lemma 31: Aus $\langle c,\ \sigma\rangle \Downarrow \sigma'$ folgt $P_1 +\!+ \text{comp}(c) +\!+ P_2 \vdash \langle |P_1|,\ \sigma\rangle \xrightarrow{*} \langle |P_1| + |\text{comp}(c)|,\ \sigma'\rangle$.
Beweis durch Regelinduktion über $\langle c,\ \sigma\rangle \Downarrow \sigma'$, P_1 und P_2 beliebig:

- Fall RepeatTT:
  Induktionsannahmen: Es gilt $\langle c,\ \sigma\rangle \Downarrow \sigma'$, $\mathcal{B}[\![b]\!]\,\sigma' = \mathbf{tt}$ und für beliebige P_1 und P_2 gilt

  $$P_1 +\!+ \text{comp}(c) +\!+ P_2 \vdash \langle |P_1|,\ \sigma\rangle \xrightarrow{*} \langle |P_1| + |\text{comp}(c)|,\ \sigma'\rangle.$$

  Zu zeigen: Für alle P_1 und P_2 gilt

  $$P_1 +\!+ \text{comp}(c) +\!+ [\texttt{JMPF}\ -|\text{comp}(c)|\ b] +\!+ P_2 \vdash \langle |P_1|,\ \sigma\rangle \xrightarrow{*} \langle |P_1| + |\text{comp}(c)| + 1,\ \sigma'\rangle.$$

  Beweis: Die Aussage folgt aus der Induktionsannahme (instanziiere dabei P_1 mit P_1 und P_2 mit $[\texttt{JMPF}\ -|\text{comp}(c)|\ b] +\!+ P_2$) gefolgt von der Regel JmpFF:

  $$P_1 +\!+ \text{comp}(c) +\!+ [\texttt{JMPF}\ -|\text{comp}(c)|\ b] +\!+ P_2 \vdash \langle |P_1|,\ \sigma\rangle \xrightarrow{*} \langle |P_1| + |\text{comp}(c)|,\ \sigma'\rangle \to \langle |P_1| + |\text{comp}(c)| + 1,\ \sigma'\rangle$$

- Fall RepeatFF: Induktionsannahmen: Es gilt $\langle c,\ \sigma\rangle \Downarrow \sigma'$, $\mathcal{B}[\![b]\!]\,\sigma' = \mathbf{ff}$,
  für beliebige P_1 und P_2 gilt

  $$P_1 +\!+ \text{comp}(c) +\!+ P_2 \vdash \langle |P_1|,\ \sigma\rangle \xrightarrow{*} \langle |P_1| + |\text{comp}(c)|,\ \sigma'\rangle$$

  weiter gilt $\langle \texttt{repeat}\ c\ \texttt{until}\ b,\ \sigma'\rangle \Downarrow \sigma''$ und für beliebige P_1 und P_2 gilt

  $$P_1 +\!+ \text{comp}(c) +\!+ [\texttt{JMPF}\ -|\text{comp}(c)|\ b] +\!+ P_2 \vdash \langle |P_1|,\ \sigma'\rangle \xrightarrow{*} \langle |P_1| + |\text{comp}(c)| + 1,\ \sigma''\rangle.$$

  Zu zeigen ist: Für alle P_1 und P_2 gilt

  $$P_1 +\!+ \text{comp}(c) +\!+ [\texttt{JMPF}\ -|\text{comp}(c)|\ b] +\!+ P_2 \vdash \langle |P_1|,\ \sigma\rangle \xrightarrow{*} \langle |P_1| + |\text{comp}(c)| + 1,\ \sigma''\rangle.$$

  Beweis:

  $$P_1 +\!+ \text{comp}(c) +\!+ [\texttt{JMPF}\ -|\text{comp}(c)|\ b] +\!+ P_2 \vdash \langle |P_1|,\ \sigma\rangle \xrightarrow{*} \langle |P_1| + |\text{comp}(c)|,\ \sigma'\rangle \to \langle |P_1|,\ \sigma'\rangle \to \langle |P_1| + |\text{comp}(c)| + 1,\ \sigma''\rangle$$

Thm. 37 Wenn $\text{comp}(c) \vdash \langle 0,\ \sigma\rangle \xrightarrow{*} \langle |\text{comp}(c)|,\ \sigma'\rangle$, dann $\langle c,\ \sigma\rangle \Downarrow \sigma'$.
Der bisherige Beweis macht eine Induktion über c. Hier muss noch folgender Fall hinzugefügt werden:

- Fall `repeat` c `until` b: Abkürzungen: $P = \text{comp}(\texttt{repeat}\ c\ \texttt{until}\ b), l = |\text{comp}(c)|$.
  Induktionsannahme: (i) Für alle σ und σ' gilt, wenn $\text{comp}(c) \vdash \langle 0,\ \sigma\rangle \xrightarrow{*} \langle l,\ \sigma'\rangle$, dann $\langle c,\ \sigma\rangle \Downarrow \sigma'$.
  Zu zeigen: Wenn $P \vdash \langle 0,\ \sigma\rangle \xrightarrow{*} \langle l+1,\ \sigma'\rangle$, dann $\langle \texttt{repeat}\ c\ \texttt{until}\ b,\ \sigma\rangle \Downarrow \sigma'$. Wir beweisen dies durch vollständige Induktion über die Länge n dieser Ableitung:
  Induktionsannahme II: Für alle $m < n$ und σ gilt: Wenn $P \vdash \langle 0,\ \sigma\rangle \xrightarrow{m} \langle l+1,\ \sigma'\rangle$, dann $\langle \texttt{repeat}\ c\ \texttt{until}\ b,\ \sigma\rangle \Downarrow \sigma'$.
  Zu zeigen: Angenommen $P \vdash \langle 0,\ \sigma\rangle \xrightarrow{n} \langle l+1,\ \sigma'\rangle$, dann $\langle \texttt{repeat}\ c\ \texttt{until}\ b,\ \sigma\rangle \Downarrow \sigma'$.
  Aus dieser Annahme folgt mit dem Aufteilungslemma 36 (da P mit comp(c) anfängt und dieses abgeschlossen ist), dass es σ^*, n_1 und n_2 gibt mit $n = n_1 + n_2$, (ii) $\text{comp}(c) \vdash \langle 0,\ \sigma\rangle \xrightarrow{n_1} \langle l,\ \sigma^*\rangle$ und (iii) $P \vdash \langle l,\ \sigma^*\rangle \xrightarrow{n_2} \langle l+1,\ \sigma'\rangle$.
  Aus (ii) folgt mit (i), dass $\langle c,\ \sigma\rangle \Downarrow \sigma^*$.
  Fallunterscheidung nach $\mathcal{B}[\![b]\!]\,\sigma^*$:

– Fall $\mathcal{B}[\![b]\!]\,\sigma^* = \mathbf{tt}$: Aus (iii) folgt dann $n_2 = 1$, $\sigma^* = \sigma'$ durch Regelinversion. Nach Regel RepeatTT gilt $\langle \texttt{repeat}\ c\ \texttt{until}\ b,\ \sigma\rangle \Downarrow \sigma'$.
– Fall $\mathcal{B}[\![b]\!]\,\sigma^* = \mathbf{ff}$: Aus (iii) folgt damit $P \vdash \langle l,\ \sigma^*\rangle \to \langle 0,\ \sigma^*\rangle \overset{n_2-1}{\to} \langle l+1,\ \sigma'\rangle$. Mit der I.A. II ($m = n_2 - 1 < n$ und $\sigma = \sigma^*$) folgt $\langle \texttt{repeat}\ c\ \texttt{until}\ b,\ \sigma^*\rangle \Downarrow \sigma'$. Zusammen mit $\mathcal{B}[\![b]\!]\,\sigma^* = \mathbf{ff}$ und $\langle c,\ \sigma\rangle \Downarrow \sigma^*$ folgt $\langle \texttt{repeat}\ c\ \texttt{until}\ b,\ \sigma\rangle \Downarrow \sigma'$ nach Regel RepeatFF.

3. Compiler für arithmetische Ausdrücke (Ü)

Für die arithmetischen Ausdrücke Aexp der Sprache While soll ein Compiler angegeben werden. Der Compiler nimmt einen Ausdruck und übersetzt ihn in eine Liste von Instruktionen einer Stackmaschine. Auf der Stackmaschine stehen folgende Instruktionen zur Verfügung:

`CONST` n Lege die Konstante n auf den Stack ($n \in \mathbb{Z}$).

`LOAD` x Lade den Wert der Variablen x auf den Stack.

`APPLY` f Wende die binäre Funktion f auf die beiden obersten Stackelemente an und ersetze sie durch das Ergebnis.

Die Maschine, auf der die übersetzten Ausdrücke abgearbeitet werden sollen, nimmt eine Liste von Instruktionen, einen (anfangs leeren) Stack und einen Zustand mit der aktuellen Variablenbelegung als Argumente. Auf die Variablenbelegung kann mit Hilfe von `LOAD` $\cdot$ zugegriffen werden. Die Stackmaschine liefert einen neuen Stack als Ergebnis, der am Ende der Berechnung nur noch deren Ergebnis enthalten soll.

(a) Geben Sie eine formale Semantik für die Stack-Maschine an.
(b) Schreiben Sie einen Compiler für arithmetische Ausdrücke.
(c) Formulieren Sie die Aussage, dass der Compiler korrekt ist, formal.
(d) Beweisen Sie die Korrektheitsaussage.

Lösung:

(3a) Modellierung:
- Zustand wird nicht verändert, also vor $\vdash$ setzen.
- Statt eines Instruktionszeigers wie bei ASM kann hier direkt die Liste der Instruktionen abgearbeitet werden, es gibt keine Sprünge.
- Variablenkonventionen: I für Instruktionsliste, s für Stack (Listendarstellung)

Abkürzung: $x \cdot xs = [x] +\!+ xs$
Semantik: $\sigma \vdash \langle I,\ s\rangle \to \langle I',\ s'\rangle$

$$\textsc{Const}:\ \sigma \vdash \langle \texttt{CONST}\ n \cdot I,\ s\rangle \to \langle I,\ n \cdot s\rangle$$

$$\textsc{Load}:\ \sigma \vdash \langle \texttt{LOAD}\ x \cdot I,\ s\rangle \to \langle I,\ \sigma(x) \cdot s\rangle$$

$$\textsc{Apply}:\ \sigma \vdash \langle \texttt{APPLY}\ f \cdot I,\ n_2 \cdot n_1 \cdot s\rangle \to \langle I,\ f(n_1, n_2) \cdot s\rangle$$

Endzustand: $I = []$, $s = [n]$.

(3b) Definition des Compilers C rekursiv über Aexp:

$$\begin{aligned} \mathrm{C}(n) &= [\texttt{CONST}\ \mathcal{N}[\![n]\!]] \\ \mathrm{C}(x) &= [\texttt{LOAD}\ x] \\ \mathrm{C}(a_1\ \texttt{-}\ a_2) &= \mathrm{C}(a_1) +\!+ \mathrm{C}(a_2) +\!+ [\texttt{APPLY}\ -] \\ \mathrm{C}(a_1\ \texttt{*}\ a_2) &= \mathrm{C}(a_1) +\!+ \mathrm{C}(a_2) +\!+ [\texttt{APPLY}\ *] \end{aligned}$$

Wichtig ist, dass die Reihenfolge der Argumente für $-$ und $*$ mit der Semantikdefinition übereinstimmt, da $-$ nicht kommutativ ist. Für einen Stack $[n_1, n_2, \ldots]$ bedeutet die Anwendung von `APPLY` $-$, dass n_1 und n_2 vom Stack genommen werden und $n_2 - n_1$ auf den Ergebnisstack gelegt werden.

(3c) Korrektheitsaussagen:

i. Für alle a und σ gilt: $\sigma \vdash \langle \mathrm{C}(a),\ [] \rangle \xrightarrow{*} \langle [],\ [\mathcal{A}\,[\![a]\!]\,\sigma] \rangle$.

ii. Für alle a, σ, und s' gilt: Wenn $\sigma \vdash \langle \mathrm{C}(a),\ [] \rangle \xrightarrow{*} \langle [],\ s' \rangle$, dann $s' = [\mathcal{A}\,[\![a]\!]\,\sigma]$.

(3d) Die erste Behauptung benötigt folgendes Lifting-Lemma:

Lemma 1. Wenn $\sigma \vdash \langle I,\ s \rangle \xrightarrow{n} \langle I',\ s' \rangle$, dann $\sigma \vdash \langle I +\!+ I^*,\ s +\!+ s^* \rangle \xrightarrow{n} \langle I' +\!+ I^*,\ s' +\!+ s^* \rangle$.

Beweis. Induktion über n (s und I beliebig):

- Fall $n = 0$: Zu zeigen: Wenn (i) $\sigma \vdash \langle I,\ s \rangle \xrightarrow{0} \langle I',\ s' \rangle$, dann $\sigma \vdash \langle I +\!+ I^*,\ s +\!+ s^* \rangle \xrightarrow{0} \langle I' +\!+ I^*,\ s' +\!+ s^* \rangle$.
  Aus (i) folgt $I' = I$ und $s' = s$ durch Regelinversion. Damit ist die Behauptung trivial.
- Fall $n + 1$: Induktionsannahme: Wenn $\sigma \vdash \langle I,\ s \rangle \xrightarrow{n} \langle I',\ s' \rangle$, dann $\sigma \vdash \langle I +\!+ I^*,\ s +\!+ s^* \rangle \xrightarrow{n} \langle I' +\!+ I^*,\ s' +\!+ s^* \rangle$.
  Zu zeigen: Wenn (i) $\sigma \vdash \langle I,\ s \rangle \xrightarrow{n+1} \langle I',\ s' \rangle$, dann $\sigma \vdash \langle I +\!+ I^*,\ s +\!+ s^* \rangle \xrightarrow{n+1} \langle I' +\!+ I^*,\ s' +\!+ s^* \rangle$.
  Aus (i) erhält man I'' und s'' mit $\sigma \vdash \langle I,\ s \rangle \to \langle I'',\ s'' \rangle \xrightarrow{n} \langle I',\ s' \rangle$.
  Aus $\sigma \vdash \langle I,\ s \rangle \to \langle I'',\ s'' \rangle$ folgt $\sigma \vdash \langle I +\!+ I^*,\ s +\!+ s^* \rangle \to \langle I'' +\!+ I^*,\ s'' +\!+ s^* \rangle$ per Fallunterscheidung nach Const, Load bzw. Apply. Mit der Induktionsannahme für I'' und s'' sowie $\sigma \vdash \langle I'',\ s'' \rangle \xrightarrow{n} \langle I',\ s' \rangle$ folgt die Behauptung. □

Beweis der ersten Behauptung durch Induktion über a:

- Fall $a = n$: Zu zeigen: $\sigma \vdash \langle [\texttt{CONST}\ n],\ [] \rangle \xrightarrow{*} \langle [],\ [n] \rangle$. Trivial mit Regel Const.
- Fall $a = x$: Zu zeigen: $\sigma \vdash \langle [\texttt{LOAD}\ x],\ [] \rangle \xrightarrow{*} \langle [],\ [\sigma(x)] \rangle$. Trivial mit Regel Load.
- Fall $a = a_1\ \texttt{-}\ a_2$: Induktionsannahmen: (i) $\sigma \vdash \langle \mathrm{C}(a_1),\ [] \rangle \xrightarrow{*} \langle [],\ [\mathcal{A}\,[\![a_1]\!]\,\sigma] \rangle$ und (ii) $\sigma \vdash \langle \mathrm{C}(a_2),\ [] \rangle \xrightarrow{*} \langle [],\ [\mathcal{A}\,[\![a_2]\!]\,\sigma] \rangle$.
  Zu zeigen: $\sigma \vdash \langle \mathrm{C}(a_1) +\!+ \mathrm{C}(a_2) +\!+ [\texttt{APPLY}\ -],\ [] \rangle \xrightarrow{*} \langle [],\ \mathcal{A}\,[\![a_1]\!]\,\sigma - \mathcal{A}\,[\![a_2]\!]\,\sigma \rangle$.
  Aus (i) und (ii) folgt mit obigem Lemma und Regel Apply:

$$\sigma \vdash \langle \mathrm{C}(a_1) +\!+ (\mathrm{C}(a_2) +\!+ [\texttt{APPLY}\ -]),\ [] \rangle \xrightarrow{*} \langle \mathrm{C}(a_2) +\!+ [\texttt{APPLY}\ -],\ [\mathcal{A}\,[\![a_1]\!]\,\sigma] \rangle \xrightarrow{*} \langle [\texttt{APPLY}\ -],\ [\mathcal{A}\,[\![a_2]\!]\,\sigma] +\!+ [\mathcal{A}\,[\![a_1]\!]\,\sigma] \rangle \to \langle [],\ \mathcal{A}\,[\![a_1]\!]\,\sigma - \mathcal{A}\,[\![a_2]\!]\,\sigma \rangle$$

- Fall $a = a_1\ \texttt{*}\ a_2$: Analog zu $a_1\ \texttt{-}\ a_2$. □

Man kann die erste Behauptung auch ohne Liftinglemma beweisen, wenn man sie schon vor der Induktion verallgemeinert zu: Für alle a, σ, I und s gilt $\sigma \vdash \langle \mathrm{C}(a) \cdot I,\ s \rangle \xrightarrow{*} \langle I,\ \mathcal{A}\,[\![a]\!]\,\sigma \cdot s \rangle$.

Beweis der zweiten Behauptung. Dies könnte man analog zum Simulationstheorem 37 mit Induktion über a zeigen, das erfordert aber entsprechende Aufteilungslemmata. Einfacher ist der Beweis über Determinismus:
Wenn $\sigma \vdash \langle I,\ s \rangle \xrightarrow{*} \langle [],\ s' \rangle$ und $\sigma \vdash \langle I,\ s \rangle \xrightarrow{*} \langle [],\ s'' \rangle$, dann $s' = s''$.
Beweis durch Induktion über I (s beliebig):

- Fall $I = []$: Aus $\sigma \vdash \langle [],\ s \rangle \xrightarrow{*} \langle [],\ s' \rangle$ folgt $s = s'$, analog folgt aus $\sigma \vdash \langle [],\ s \rangle \xrightarrow{*} \langle [],\ s'' \rangle$ dass $s = s''$, also gilt $s = s''$.
- Fall $i \cdot I$: (i ist eine Instruktion). Gegeben sind (i) $\sigma \vdash \langle i \cdot I,\ s \rangle \xrightarrow{*} \langle [],\ s' \rangle$ und (ii) $\sigma \vdash \langle i \cdot I,\ s \rangle \xrightarrow{*} \langle [],\ s'' \rangle$.
  Da $i \cdot I \neq []$, erhält man aus (i) dass es I'^* und s'^* gibt mit $\sigma \vdash \langle i \cdot I,\ s \rangle \to \langle I'^*,\ s'^* \rangle \xrightarrow{*} \langle [],\ s' \rangle$.
  Analog erhält man aus (ii), dass es I''^* und s''^* gibt mit $\sigma \vdash \langle i \cdot I,\ s \rangle \to \langle I''^*,\ s''^* \rangle \xrightarrow{*} \langle [],\ s'' \rangle$.

Durch Regelinversion folgt nun dass $I''^* = I'^* = I$ und $s'^* = s''^*$ (Ein-Schrittt-Determinismus). Damit folgt die Behauptung mit der Induktionsannahme aus $\sigma \vdash \langle I,\ s'^* \rangle \xrightarrow{*} \langle [],\ s' \rangle$ und $\sigma \vdash \langle I,\ s'^* \rangle \xrightarrow{*} \langle [],\ s'' \rangle$.

Die Behauptung folgt dann aus dem Determinismus zusammen mit der vorherigen Behauptung. □

Anmerkung: Der Beweis über den Determinismus funktioniert nur, weil die erste Behauptung bereits beinhaltet, dass alle Ausführungen der Stack-Maschine terminieren. Für die Korrektheit von comp von While nach ASM hätte dies nicht funktioniert.