Lehrstuhl für Programmierparadigmen

Semantik von Programmiersprachen

Sommersemester 2017

Andreas Lochbihler

überarbeitet von

Joachim Breitner, Denis Lohner und Sebastian Ullrich
{breitner,denis.lohner,sebastian.ullrich}@kit.edu

Stand
24. Juli 2017

Inhaltsverzeichnis

Organisatorisches

Termine

Vorlesung	2-stündig	Di, 11.30 – 13.00h,	SR 301, Informatik-Hauptgebäude
Übung:	2-stündig	Mo, 9.45 – 11.15h,	SR -118, Informatik-Hauptgebäude

Zum ersten Übungstermin findet eine Vorlesung statt.

Unterlagen

Skript und Übungsblätter auf
`http://pp.ipd.kit.edu/lehre/SS2017/semantik/`

Übungsblätter:

- Veröffentlichung am Dienstag nach der Vorlesung
- Besprechung in der/einer folgenden Übung
- Keine Abgabe und keine Korrektur

Anrechenbarkeit

Master Informatik (SPO 2008 & 2015):
Modul: Semantik von Programmiersprachen [M-INFO-100845]
ECTS-Punkte: 4

Literatur

- Hanne Riis Nielson, Flemming Nielson: Semantics with Applications. An Appetizer.
  Springer, 2007. ISBN: 978-1-84628-691-9.
  Grundlage der meisten Themen der Vorlesung, sehr anschaulich und gut verständlich
- Benjamin C. Pierce: Types and Programming Languages.
  MIT Press, 2002. ISBN: 0-262-162209-1.
  Schwerpunkt auf dem Lambda-Kalkül und Typsystemen, mit sehr guten Erklärungen, auch zu weiterführenden Themen.
- Glynn Winskel: The Formal Semantics of Programming Languages. An Introduction.
  MIT Press, 1993. ISBN: 0-262-23169-7.
  Ausführlicher Beweis der Unentscheidbarkeit eines vollständigen axiomatischen Kalküls
- Samson Abramsky, C.H. Luke Ong: Full Abstraction in the Lazy Lambda Calculus
  in Information and Computation, Elsevier, 1993.
  Grundlage für das Kapitel zum Lambda-Kalkül.
- Tobias Nipkow, Gerwin Klein: Concrete Semantics
  Springer, 2014. ISBN: 978-3-319-10542-0.
  `http://www.concrete-semantics.org/`
  Formalisierung des Vorlesungsstoffs im Theorembeweiser Isabelle, inklusive einer Einführung in diesen.

1 Einführung

1.1 Was ist eine Semantik?

Syntax und Semantik sind zwei unabdingbare Bausteine der Beschreibung von (Programmier-)Sprachen: Syntax kümmert sich darum, welche Zeichenfolgen gültige Sätze (Programme) der Sprache sind. Syntax umfasst also Vokabular (Schlüsselwörter) und Grammatik. Semantik beschreibt, was die Bedeutung eines gültigen Satzes (Programms) sein soll. Für Programmiersprachen heißt das: Wie verhält sich das Programm, wenn man es ausführt?

Syntax legt auch den Aufbau eines Satzes, i. d. R. ein Baum, fest und erklärt wie man von einer Zeichenfolge zum Syntaxbaum kommt. Eine Semantik beschreibt, wie man dieser syntaktischen Struktur eine Bedeutung gibt, d.h.: Was ist die Bedeutung eines einzelnen Konstrukts? Wie erhält man die Gesamtbedeutung aus den einzelnen Teilen?

Syntax und Semantik vieler Programmiersprachen sind standardisiert (C, C++, Pascal, Java, ...). Für die Definition der Syntax werden formale Techniken routinemäßig in der Praxis eingesetzt: kontextfreie Grammatiken (EBNF). Das Verhalten von Konstrukten der Programmiersprache und deren Zusammenwirken beschreiben die meisten dieser Standards allerdings nur in natürlicher Sprache, meistens auf englisch oder nur anhand konkreter Beispiele. Für umfangreiche Programmiersprachen ist es auf diese Weise fast unmöglich, alle möglichen Kombinationen eindeutig festzulegen und dabei trotzdem Widerspruchsfreiheit zu garantieren. Deswegen gibt es auch formale, d.h. mathematische, Beschreibungstechniken für Semantik, die das Thema dieser Vorlesung sind. Die funktionale Sprache ML wurde beispielsweise vollständig durch eine formale Semantik definiert; auch für Java gibt es formale Modellierungen, die zwar nicht Teil der Sprachdefinition sind, aber den Entwurf und die Weiterentwicklungen wesentlich beeinflussten (z.B. Featherweight Java, Java light, Jinja).

1.2 Warum formale Semantik?

Die einfachste Definition einer Sprache ist mittels eines Compilers: Alle Programme, die der Compiler akzeptiert, sind syntaktisch korrekt; die Semantik ist die des Zielprogramms. Eine solche Definition ist aber sehr problematisch:

1. *Keine Abstraktion.* Um das Verhalten eines Programmes zu verstehen, muss man den Compiler und das Kompilat in der Zielsprache verstehen. Für neue, abstrakte Konzepte in der Programmiersprache gibt es keine Garantie, dass die Abstraktionsschicht nicht durch andere Konstrukte durchbrochen werden kann – geht es doch, ist das eine der Hauptfehlerquellen bei der Entwicklung von Programmen.

   Beispiele:

   - Pointer-Arithmetik in C++ kann die Integrität von Objekten zerstören, weil damit beliebig (auch auf private) Felder zugegriffen werden kann.
   - `setjmp/longjmp` in C widerspricht dem Paradigma, dass Methoden stack-artig aufgerufen werden.
   - LaTeX ist nur ein Aufsatz auf TeX, der zwar in vielen Fällen eine gute Abstraktionsschicht schafft, aber diese nicht garantieren kann. Fast jeder ist schon über unverständliche TeX-Fehlermeldungen und Inkompatibilitäten zwischen LaTeX-Paketen gestoplert.

2. *Plattformabhängigkeit und Überspezifikation.* Ein Wechsel auf eine andere Zielsprache oder einen anderen Compiler ist fast unmöglich. Schließlich ist auch festgelegt, wie viele einzelne Aus-

führungsschritte (z. B. Anzahl der Prozessorzyklen) jedes einzelne Konstrukt genau benötigen muss.

Zwischen „Sprachdefinition“ und Implementierungsdetails des Compilers kann nicht unterschieden werden. Weiterentwicklungen aus der Compiler-Technik sind damit ausgeschlossen.

3. *Bootstrapping.* Auch ein Compiler ist nur durch seinen Programmtext definiert. Was ist aber die Semantik dieses Programmtextes?

Eine Semantikbeschreibung in Prosa wie bei den meisten Sprachstandards ist zwar besser, kann aber Mehrdeutigkeiten, Missverständnisse oder Widersprüche auch nicht verhindern. Demgegenüber stehen die Vorteile mathematischer Beschreibungen einer Semantik:

1. *Vollständige, rigorose Definition einer Sprache.* Jedes syntaktisch gültige Programm hat eine eindeutige, klar festgelegte Semantik. Mathematische Beschreibungen sind syntaktisch oft viel kürzer als englischer Fließtext. Programmierer können die Semantik als Nachschlagereferenz verwenden, um subtile Feinheiten der Sprache zu verstehen. Compiler-Bauer können die Semantik einer solchen Sprache als Korrektheitskriterium ihres Compilers verwenden. Damit verhalten sich Anwender-Programme gleich, unabhängig vom verwendeten (korrekten) Compiler.

2. *Nachweis von Programmeigenschaften.* Ohne formale Semantik als Grundlage lassen sich Eigenschaften eines Programms nicht beweisen, ja nicht einmal mathematisch aufschreiben. Dabei unterscheidet man zwischen Eigenschaften, die alle Programme erfüllen und damit Meta-Eigenschaften der Sprache bzw. Semantik sind (z. B. Typsicherheit), und solchen eines einzelnen Programms (z. B. Korrektheit eines Algorithmus).

3. *Unterstützung beim Programmiersprachenentwurf.* Eine Programmiersprache mit vielen verschiedenen Konzepten, die klar, verständlich und ohne unnötige Sonderfälle zusammenwirken, zu entwerfen, ist sehr schwierig. Bereits der Versuch, eine formale Semantik für eine Programmiersprache zu entwerfen, deckt viele Inkonsistenzen und unerwartete Interaktionen auf.

   Hat man eine formale Beschreibung der Semantik, lässt sich automatisch ein prototypischer Interpreter für die Sprache erzeugen, z. B. als Prolog-Programm. Dadurch können verschiedene Designentscheidungen beim Entwurf der Semantik an konkreten Beispielen praktisch ausprobiert werden.

   Programmverifikation ist einfacher, wenn die Semantik der Programmiersprache mathematisch einfach und klar ist. Eine zufällig, nach Gutdünken entworfene Programmiersprache hat in der Regel ein sehr kompliziertes mathematisches Modell. Dementsprechend ist es auch schwierig, darüber Beweise zu führen, da stets viele Sonderfälle berücksichtigt werden müssen.

4. *Klare und konsistente Begrifflichkeit.* Formale Semantik arbeitet mit der Sprache und den Begriffen der Mathematik, über deren Bedeutung man sich im Klaren ist. Damit werden Mehrdeutigkeiten und Missverständnisse von vornherein ausgeschlossen.

Beispiele:

- Was ist der Wert von `x` am Ende?

```
b = true; c = false; x = 0;
if (b) if (c) x = 1; else x = 2;
```

- Was ist der Wert von `i` nach Ausführung des folgenden C-Fragments? Was wäre er in Java?

```
int i = 1;
i += i++ + ++i;
```

- Sind die beiden Initialisierungen von `b` äquivalent?

```
boolean f(int a, int b) {
  return (a == 0) && (b == 0);
}

boolean b = (1 == 0) && (2 / 0 == 0);
boolean b = f(1, 2 / 0);
```

1.3 Operational, denotational und axiomatisch

In dieser Vorlesung werden die drei bekanntesten Beschreibungsarten für Semantiken vorgestellt – mit einem Schwerpunkt auf operationaler Semantik:

Operational Die Bedeutung eines Konstrukts ergibt sich aus seinen (abstrakten) Ausführungsschritten, die durch symbolische Regeln beschrieben werden. Neben dem Ergebnis der Berechnung wird auch modelliert, wie die Berechnung zum Ergebnis kommt. Die Regeln beschreiben dabei nur eine mögliche (idealisierte) Implementierung der Sprache, reale Implementierungen können andere, äquivalente Abläufe verwenden.

Denotational Jedes Konstrukt wird als mathematisches Objekt realisiert, welches *nur* den Effekt seiner Ausführung modelliert. Im Gegensatz zur operationalen Semantik ist irrelevant, wie der Effekt zustande kommt.

Axiomatisch Die Bedeutung eines Programms wird indirekt festgelegt, indem beschrieben wird, welche Eigenschaften des Programms (in einem zu entwerfenden Logik-Kalkül) beweisbar sind.

Die drei verschiedenen Ansätze ergänzen sich: Für Compiler und Implementierungen sind operationale Semantiken gut geeignet. Denotationale Konzepte finden sich oft in der Programmanalyse. Programmverifikation stützt sich auf die axiomatische Semantik. Zu jedem Ansatz werden wir eine Semantik für eine einfache imperative Sprache – ggf. mit verschiedenen Spracherweiterungen – entwickeln und die Beziehung der verschiedenen Semantiken zueinander untersuchen.

Beispiel 1. Semantik des Programms `z := x; x := y; y := z`

1. Die *operationale Semantik* beschreibt die Semantik, indem sie definiert, wie das Programm auszuführen ist: Eine Sequenz von zwei durch `;` getrennten Anweisungen führt die einzelnen Anweisungen nacheinander aus. Eine Zuweisung der Form $\langle \mathit{Variable} \rangle$ `:=` $\langle \mathit{Ausdruck} \rangle$ wertet zuerst den Ausdruck zu einem Wert aus und weist diesen Wert dann der Variablen zu.

   Für einen Zustand $[\mathtt{x} \mapsto 5, \mathtt{y} \mapsto 7, \mathtt{z} \mapsto 0]$, der den Variablen `x`, `y` und `z` die Werte 5, 7 und 0 zuweist, ergibt sich folgende Auswertungssequenz:

$$
\begin{aligned}
& \langle \mathtt{z := x;\ x := y;\ y := z},\ [\mathtt{x} \mapsto 5, \mathtt{y} \mapsto 7, \mathtt{z} \mapsto 0] \rangle \\
\to_1\ & \langle \mathtt{x := y;\ y := z},\ [\mathtt{x} \mapsto 5, \mathtt{y} \mapsto 7, \mathtt{z} \mapsto 5] \rangle \\
\to_1\ & \langle \mathtt{y := z},\ [\mathtt{x} \mapsto 7, \mathtt{y} \mapsto 7, \mathtt{z} \mapsto 5] \rangle \\
\to_1\ & [\mathtt{x} \mapsto 7, \mathtt{y} \mapsto 5, \mathtt{z} \mapsto 5]
\end{aligned}
$$

2. Die *denotationale Semantik* kümmert sich nur um den Effekt der Ausführung, nicht um die einzelnen Berechnungsschritte. Entsprechend ist die Semantik eines solchen Programms eine

Funktion, die einen Ausgangszustand in einen Endzustand überführt. Für eine Sequenz erhält man die Funktion durch Komposition (Hintereinanderausführung) der Funktionen der beiden Anweisungen. Die Bedeutung einer Zuweisung ist die Funktion, die den übergebenen Zustand so ändert, dass der Variablen dann der Wert des Ausdrucks zugewiesen ist.

Für das Programm ergibt sich dann:

$$\begin{aligned}
\mathcal{D}[\![\texttt{z := x; x := y; y := z}]\!] &= \mathcal{D}[\![\texttt{y := z}]\!] \circ \mathcal{D}[\![\texttt{x := y}]\!] \circ \mathcal{D}[\![\texttt{z := x}]\!] \\
&= \lambda\sigma.\, \mathcal{D}[\![\texttt{y := z}]\!]\,(\mathcal{D}[\![\texttt{x := y}]\!]\,(\mathcal{D}[\![\texttt{z := x}]\!]\,(\sigma))) \\
&= \lambda\sigma.\, \mathcal{D}[\![\texttt{y := z}]\!]\,(\mathcal{D}[\![\texttt{x := y}]\!]\,(\sigma[\texttt{z} \mapsto \sigma(\texttt{x})])) \\
&= \lambda\sigma.\, \mathcal{D}[\![\texttt{y := z}]\!]\,(\sigma[\texttt{z} \mapsto \sigma(\texttt{x}),\ \texttt{x} \mapsto \sigma[\texttt{z} \mapsto \sigma(\texttt{x})](\texttt{y})]) \\
&= \lambda\sigma.\, \mathcal{D}[\![\texttt{y := z}]\!]\,(\sigma[\texttt{z} \mapsto \sigma(\texttt{x}),\ \texttt{x} \mapsto \sigma(\texttt{y})]) \\
&= \lambda\sigma.\, \sigma[\texttt{z} \mapsto \sigma(\texttt{x}),\ \texttt{x} \mapsto \sigma(\texttt{y}),\ \texttt{y} \mapsto \sigma[\texttt{z} \mapsto \sigma(\texttt{x}),\ \texttt{y} \mapsto \sigma(\texttt{y})](\texttt{z})] \\
&= \lambda\sigma.\, \sigma[\texttt{x} \mapsto \sigma(\texttt{y}),\ \texttt{y} \mapsto \sigma(\texttt{x}),\ \texttt{z} \mapsto \sigma(\texttt{x})]
\end{aligned}$$

Für $\sigma = [\texttt{x} \mapsto 5, \texttt{y} \mapsto 7, \texttt{z} \mapsto 0]$ ergibt dies wieder:

$$\mathcal{D}[\![\texttt{z := x; x := y; y := z}]\!]\,(\sigma) = [\texttt{x} \mapsto 7, \texttt{y} \mapsto 5, \texttt{z} \mapsto 5]$$

3. *Axiomatische Semantik* konzentriert sich auf die Korrektheit eines Programms bezüglich Vor- und Nachbedingungen. Immer, wenn ein Startzustand die Vorbedingung erfüllt, dann sollte – sofern das Programm terminiert – nach der Ausführung des Programms mit diesem Startzustand der Endzustand die Nachbedingung erfüllen.

   $$\{\,\texttt{x} = n \wedge \texttt{y} = m\,\}\,\texttt{z := x; x := y; y := z}\,\{\,\texttt{x} = m \wedge \texttt{y} = n\,\}$$

   Dabei ist $\texttt{x} = n \wedge \texttt{y} = m$ die Vorbedingung und $\texttt{x} = m \wedge \texttt{y} = n$ die Nachbedingung. Die Hilfsvariablen (logical variables) n und m, die nicht im Programm vorkommen, merken sich die Anfangswerte von `x` und `y`.

   Eine axiomatische Semantik definiert ein Regelsystem, mit dem Aussagen wie obige hergeleitet werden können. Beispielsweise ist $\{\,P\,\}\,c_1\texttt{;}\ c_2\,\{\,Q\,\}$ für eine Sequenz $c_1\texttt{;}\ c_2$ herleitbar, wenn $\{\,P\,\}\,c_1\,\{\,R\,\}$ und $\{\,R\,\}\,c_2\,\{\,Q\,\}$ für eine Bedingung R herleitbar sind. Dadurch ergeben sich viele Bedingungen an das Programmverhalten, die dieses dadurch (möglichst vollständig) beschreiben. Wichtig dabei ist, dass die Regeln korrekt sind, d.h, keine widersprüchlichen Bedingungen herleitbar sind. Umgekehrt sollen sich möglichst alle Eigenschaften eines Programms durch das Kalkül herleiten lassen (*Vollständigkeit*).

2 Mathematische Grundlagen

2.1 Prädikatenlogik höherer Stufe

Die Definitionen und Beweise in diesem Skript verwenden *typisierte Prädikatenlogik höherer Stufe*, d.h. jeder Term t hat einen Typ τ (geschrieben $t :: \tau$), und die Argumente von Prädikaten sowie Funktionen dürfen selbst wieder Prädikate bzw. Funktionen sein.

Quantoren ($\forall$, $\exists$) erstrecken sich daher auch immer nur über die Werte des Typs der gebundenen Variablen. Z.B. wird in der Aussage $\forall n.\ f(n) \geq 0$ nur über die ganzen Zahlen ($n :: \mathbb{Z}$) quantifiziert.

2.1.1 Typen

Die in der Logik verwendeten Typen sind nicht leer. D.h. von jedem Typ gibt es mindestens ein Element. Dies ist nötig, um keine Inkonsistenzen in die Logik einzuführen – wäre die Aussage $\exists x.x = x$ für den leeren Typ gültig?

Im Rahmen dieser Vorlesung werden folgende Typen verwendet:

- $\mathbb{Z}$: ganze Zahlen
- $\mathbb{B} = \{\mathbf{tt}, \mathbf{ff}\}$: Wahrheitswerte
- $\alpha + \beta$: Summentyp – alle Elemente aus α und β
- $\alpha \times \beta$: Tupeltyp – alle Elemente (a, b) mit $a :: \alpha$ und $b :: \beta$
- $\alpha \Rightarrow \beta$: Funktionstyp – alle Funktionen mit Definitionsbereich α und Wertebereich β
- $[\alpha]$: Listentyp – alle (endlichen) Listen mit Elementen aus α
- *algebraische Datentypen* – meist implizit durch kontext-freie BNF Grammatik gegeben (jede Produktion entspricht einem Konstruktor des Datentyps)

2.1.2 Notation

Typen werden manchmal als Menge aufgefasst (z.B. $123 \in \mathbb{Z}$), Prädikate als Teilmenge des entsprechenden Typs ($even \subseteq \mathbb{Z}$). Die Konvention $\alpha \equiv \beta$ führt α als abkürzende Schreibweise des Typs β ein (z.B. $\Sigma \equiv \mathsf{Var} \Rightarrow \mathbb{Z}$).

Das *Funktionsupdate* $f[x \mapsto y]$ ist definiert durch

$$f[x \mapsto y](x') = \begin{cases} y & \text{falls } x' = x \\ f(x') & \text{sonst.} \end{cases}$$

Die *Konkatenation* der Listen xs und ys wird mit $xs +\!\!+ ys$ notiert.

2.2 Induktive Prädikate

Fast alle Semantiken, die in dieser Vorlesung behandelt werden, werden durch *induktive Prädikate* definiert.

Ein induktives Prädikat $p \subseteq \alpha$ ist durch eine Menge von *Inferenzregeln* gegeben. Jede Inferenzregel hat beliebig viele Prämissen und eine Konklusion. Der Term der Konklusion muss dabei mit p beginnen. Die Regeln, bei denen p nicht in den Prämissen vorkommt, heißen *Basisregeln.*

Formal ist p dann definiert als die kleinste Teilmenge von α, die unter den gegebenen Regeln abgeschlossen ist.

Dadurch ergibt sich auch immer ein Induktionsprinzip für p. Ein Beweis nach diesem Prinzip wird als *Regelinduktion* oder einfach mit *Induktion nach* p bezeichnet.

Mit Hilfe eines *Ableitungsbaums* lassen sich erfüllende Belegungen $p(x)$ nachweisen.

Ist in einem Beweis eine erfüllende Belegung $p(x)$ gegeben, kann man auch Fallunterscheidung nach den Inferenzregeln von p machen. Dieses Beweisprinzip nennt man *Regelinversion.*

Beispiel 2. Das induktive Prädikat $sorted \subseteq [\mathbb{Z}]$ für sortierte Listen ganzer Zahlen sei definiert durch die folgenden Regeln:

$$\textsc{Nil}_{\textsc{sorted}}\colon\ sorted([]) \qquad \textsc{Single}_{\textsc{sorted}}\colon\ sorted([n])$$

$$\textsc{Cons}_{\textsc{sorted}}\colon\ \frac{n \leq m \qquad sorted([m] +\!\!+ ns)}{sorted([n, m] +\!\!+ ns)}$$

Der Ableitungsbaum für $sorted([1, 5, 7])$ ist dann

$$\dfrac{1 \leq 5 \qquad \dfrac{5 \leq 7 \qquad \dfrac{}{sorted([7])}\ \textsc{Single}_{\textsc{sorted}}}{sorted([5, 7])}\ \textsc{Cons}_{\textsc{sorted}}}{sorted([1, 5, 7])}\ \textsc{Cons}_{\textsc{sorted}}$$

Die Induktionsregel für *sorted* ist

$$\textsc{Induct}_{\textsc{sorted}}:\ \frac{\begin{array}{c} sorted(ns) \qquad P([]) \qquad \forall n.\ P([n]) \\ \forall n, m, ns.\ \ (n \leq m \wedge sorted([m] +\!\!+ ns) \wedge P([m] +\!\!+ ns)) \longrightarrow P([n, m] +\!\!+ ns) \end{array}}{P(ns)}$$

Wie kommt man darauf? Wir werden weiter unten die Korrektheit der Regel beweisen, hier sei nur das allgemeine Schema erklärt:

Eine Induktionsregel zu einem induktiven Prädikat benötigt zuerst immer eine Instanz des Prädikats (hier $sorted(ns)$) auch genannt *große Prämisse.* Die Konklusion ist dann die allgemeinste Folgerung aus dieser Prämisse: ein beliebiges Prädikat P über alle freien Variablen der Instanz, hier ns. Wir folgern die Konklusion aus dem Ableitungsbaum der großen Prämisse, indem wir zeigen, dass P in den Baumblättern gilt (Induktionsanfänge) und in den inneren Knoten erhalten bleibt (Induktionsschritte). Dazu fordern wir für jede Inferenzregel des Prädikats eine *kleine Prämisse*, in der Anwendungen des

Prädikats durch P ersetzt werden (und zusätzlich die impliziten Allquantoren der Regel explizit gemacht werden). So erhalten wir für ein Prädikat mit n Inferenzregeln eine Induktionsregel mit $n+1$ Prämissen.

Beweisregel für Regelinversion für *sorted*:

$$\frac{\forall n.\ ns = [n] \longrightarrow P' \qquad \begin{array}{c} sorted(ns) \qquad ns = [] \longrightarrow P' \\ \forall n, m, ns'.\ \big(ns = [n, m] +\!\!+ ns' \wedge n \leq m \wedge sorted([m] +\!\!+ ns')\big) \longrightarrow P' \end{array}}{P'}$$

Diese Regel lässt sich aus der Induktionsregel ableiten, indem man dort $P(ms) := (ns = ms \longrightarrow P')$ setzt und die Induktionsannahme $P([m] +\!\!+ ns)$ weglässt.

Beispielbeweis: Sei $head :: [\mathbb{Z}] \Rightarrow \mathbb{Z}$ so definiert, dass $head([n] +\!\!+ ns) = n$ gilt. Zeige: Für jede nicht leere sortierte Liste ns gilt $\forall n \in ns.\ head(ns) \leq n$.

Beweis. Induktion nach der Regel $\textsc{Induct}_{\textsc{sorted}}$:

Es ist $P(ns) = ns \neq [] \longrightarrow (\forall n \in ns.\ head(ns) \leq n)$. Nach der Induktionsregel für *sorted* ist zu zeigen:

- $P([]) = [] \neq [] \longrightarrow (\forall n \in ns.\ head(ns) \leq n)$, ist trivialerweise wahr.
- $\forall n.\ P([n]) = \forall n.\ [n] \neq [] \longrightarrow (\forall n' \in [n].\ head([n]) \leq n') = \forall n.\ \forall n' \in \{n\}.n \leq n' = \forall n.\ n \leq n$, ist wahr.
- Seien n, m, ns beliebig mit $n \leq m$, $sorted([m] +\!\!+ ns)$ und $P([m] +\!\!+ ns)$. Zu zeigen ist $P([n, m] +\!\!+ ns)$.

  Wegen der Induktionshypothese $P([m] +\!\!+ ns)$ gilt $\forall n' \in [m] +\!\!+ ns.\ m \leq n'$. Damit gilt: $P([n, m] +\!\!+ ns) = \forall n' \in [n, m] +\!\!+ ns.\ n \leq n'$.

  Sei $n' \in [n, m] +\!\!+ ns$. Fallunterscheidung nach n':

  - Fall $n' = n$: Dann gilt $n \leq n'$ trivialerweise.
  - Fall $n' \neq n$: Dann gilt $n' \in [m] +\!\!+ ns$, und mit der Induktionshypothese $m \leq n'$. Zusammen mit $n \leq m$ gilt also auch $n \leq n'$

□

Was hat es mit der kleinsten Menge, die unter den Regeln von *sorted* abgeschlossen ist, auf sich?

Betrachtet man *sorted* als Teilmenge von $[\mathbb{Z}]$, kann man die Regeln auch so schreiben:

$$\textsc{Nil}_{\textsc{sorted}}\colon [] \in sorted \qquad \textsc{Single}_{\textsc{sorted}}\colon [n] \in sorted$$

$$\textsc{Cons}_{\textsc{sorted}}\colon \frac{n \leq m \qquad [m] +\!\!+ ns \in sorted}{[n, m] +\!\!+ ns \in sorted}$$

Damit muss für jedes Prädikat p, welches diese Regeln erfüllen soll, gelten:

$$\{[]\} \cup \{[n] |\ n \in \mathbb{Z}\} \cup \{[n, m] +\!\!+ ns |\ n \leq m \wedge [m] +\!\!+ ns \in p\} \subseteq p$$

Das kleinste Prädikat, welches diese Ungleichung erfüllt, ist der Schnitt über alle Prädikate, die die Ungleichung erfüllen. Das Prädikat *sorted* ist also definiert durch:

$$sorted \equiv \bigcap \{p|\ \{[]\} \cup \{[n]|\ n \in \mathbb{Z}\} \cup \{[n,m] +\!\!+ ns|\ n \leq m \wedge [m] +\!\!+ ns \in p\} \subseteq p\}$$

Offensichtlich gelten damit die Regeln $\text{NIL}_{\text{SORTED}}$ und $\text{SINGLE}_{\text{SORTED}}$. Die Regel $\text{CONS}_{\text{SORTED}}$ lässt sich direkt beweisen:

Beweis. Sei $n \leq m$ und $[m] +\!\!+ ns \in sorted$. Z.Z. $[n,m] +\!\!+ ns \in sorted$:

$$[n,m] +\!\!+ ns \in \bigcap \big\{p|\ \{[]\} \cup \big\{[n']|\ n' \in \mathbb{Z}\big\} \cup \big\{[n',m'] +\!\!+ ns'|\ n' \leq m' \wedge [m'] +\!\!+ ns' \in p\big\} \subseteq p\big\}$$
$$\longleftrightarrow \forall p.\ \big(\{[]\} \cup \big\{[n']|\ n' \in \mathbb{Z}\big\} \cup \big\{[n',m'] +\!\!+ ns'|\ n' \leq m' \wedge [m'] +\!\!+ ns' \in p\big\} \subseteq p\big) \longrightarrow [n,m] +\!\!+ ns \in p$$

Sei also p' beliebig mit $\{[]\} \cup \{[n']|\ n' \in \mathbb{Z}\} \cup \{[n',m'] +\!\!+ ns'|\ n' \leq m' \wedge [m'] +\!\!+ ns' \in p'\} \subseteq p'$. Z.Z. ist noch $[n,m] +\!\!+ ns \in p'$.

Wegen $[m] +\!\!+ ns \in sorted$ wissen wir

$$[m] +\!\!+ ns \in \bigcap \big\{p|\ \{[]\} \cup \big\{[n']|\ n' \in \mathbb{Z}\big\} \cup \big\{[n',m'] +\!\!+ ns'|\ n' \leq m' \wedge [m'] +\!\!+ ns' \in p\big\} \subseteq p\big\}$$
$$\longleftrightarrow \forall p.\ \big(\{[]\} \cup \big\{[n']|\ n' \in \mathbb{Z}\big\} \cup \big\{[n',m'] +\!\!+ ns'|\ n' \leq m' \wedge [m'] +\!\!+ ns' \in p\big\} \subseteq p\big) \longrightarrow [m] +\!\!+ ns \in p$$

Damit folgt $[m] +\!\!+ ns \in p'$. Da $\{[n',m'] +\!\!+ ns'|\ n' \leq m' \wedge [m'] +\!\!+ ns' \in p'\} \subseteq p'$ gilt, sowie nach Annahme auch $n \leq m$, folgt daraus die zu zeigende Aussage $[n,m] +\!\!+ ns \in p'$. □

Schließlich können wir auch die Gültigkeit der Induktionsregel $\text{INDUCT}_{\text{SORTED}}$ beweisen:

Beweis. Sei ns gegeben, so dass $sorted(ns)$ gilt. D.h. es gilt:

$$ns \in \bigcap \big\{p|\ \{[]\} \cup \{[n]|\ n \in \mathbb{Z}\} \cup \big\{[n,m] +\!\!+ ns'|\ n \leq m \wedge [m] +\!\!+ ns' \in p\big\} \subseteq p\big\}$$

bzw.

$$\forall p.\ \big(\{[]\} \cup \{[n]|\ n \in \mathbb{Z}\} \cup \big\{[n,m] +\!\!+ ns'|\ n \leq m \wedge [m] +\!\!+ ns' \in p\big\} \subseteq p\big) \longrightarrow ns \in p$$

Instanziiert man in dieser Aussage für p nun $sorted \cap P$ so erhält man die zu zeigende Aussage $P(ns)$, wenn

$$\{[]\} \cup \{[n]|\ n \in \mathbb{Z}\} \cup \big\{[n,m] +\!\!+ ns'|\ n \leq m \wedge [m] +\!\!+ ns' \in (sorted \cap P)\big\} \subseteq (sorted \cap P)$$

gilt.

Dies folgt aber direkt aus den Regeln $\text{NIL}_{\text{SORTED}}$, $\text{SINGLE}_{\text{SORTED}}$ und $\text{CONS}_{\text{SORTED}}$, sowie den übrigen drei Annahmen (den kleinen Prämissen) der Induktionsregel $\text{INDUCT}_{\text{SORTED}}$.

□

3 Die Sprache While

While ist eine einfache, imperative Programmiersprache, für die in dieser Vorlesung in allen drei Ansätzen eine Semantik ausgearbeitet wird. Obwohl eine Semantik erst auf einem abstrakten Syntaxbaum aufbaut, muss man sich auf eine konkrete Syntax festlegen, um überhaupt Programme textuell aufschreiben zu können.

3.1 Syntax

Programme werden üblicherweise `in Schreibmaschinenschrift` dargestellt. Deren Syntax wird durch eine BNF-Grammatik mit drei syntaktischen Kategorien beschrieben: arithmetische Ausdrücke Aexp, boolesche Ausdrücke Bexp und Anweisungen Com. Außerdem braucht man noch numerische Literale Num und einen unendlichen Vorrat Var an (Programm-)Variablen. Die Darstellung der numerischen Literale und (Programm-)Variablen ist nicht weiter relevant, im Folgenden werden die Dezimaldarstellung (z.B. `120`, `5`) bzw. einfache Zeichenketten wie `x`, `catch22` verwendet.

Variablenkonvention: Um nicht ständig die syntaktische Kategorie einer (Meta-)Variable angeben zu müssen, bezeichne a stets einen arithmetischen Ausdruck, b einen booleschen, c eine Anweisung, n ein numerisches Literal und x eine Programmvariable aus Var – entsprechende Dekorationen mit Indizes, Strichen, usw. eingeschlossen. Dabei muss man zwischen einer Meta-Variablen x, die für eine beliebige, aber feste Programmvariable aus Var steht, und den konkreten Programmvariablen wie `x`, `y` selbst unterscheiden.

Definition 3 (Syntax von While). Die Syntax von Ausdrücken und Anweisungen sei durch folgende kontext-freie BNF-Grammatik gegeben:

$$
\begin{array}{lllll}
\text{Aexp} & a & ::= & n \mid x \mid a_1 \texttt{ - } a_2 \mid a_1 \texttt{ * } a_2 \\
\text{Bexp} & b & ::= & \texttt{true} \mid a_1 \texttt{ <= } a_2 \mid \texttt{not } b \mid b_1 \texttt{ \&\& } b_2 \\
\text{Com} & c & ::= & \texttt{skip} \mid x \texttt{ := } a \mid c_1\texttt{; } c_2 \mid \texttt{if (}b\texttt{) then } c_1 \texttt{ else } c_2 \mid \texttt{while (}b\texttt{) do } c
\end{array}
$$

Obwohl diese Sprache minimalistisch ist, sind viele weitere Programmkonstrukte ausdrückbar: Beispielsweise sind $a_1 \texttt{ + } a_2$, $a_1 \texttt{ == } a_2$, `false` und $b_1 \texttt{ || } b_2$ nur syntaktischer Zucker für

$$
a_1 \texttt{ - (0 - } a_2\texttt{)}, \qquad \texttt{(}a_1 \texttt{ <= } a_2\texttt{) \&\& (}a_2 \texttt{ <= } a_1\texttt{)}, \qquad \texttt{not true} \quad \text{und} \quad \texttt{not ((not } b_1\texttt{) \&\& (not } b_2\texttt{))}
$$

Durch diese Reduktion auf das Wesentliche wird es einfacher, eine Semantik anzugeben und Beweise zu führen, da weniger Fälle berücksichtigt werden müssen.

Obige Grammatik ist mehrdeutig, z. B. hat `while (true) do skip; x := y` zwei verschiedene Ableitungsbäume, d.h. zwei verschiedene Syntaxbäume. Da die Semantik aber erst auf dem Syntaxbaum aufbaut, ist das nicht wesentlich: Man kann immer mittels zusätzlicher Klammern den gewünschten Baum eindeutig beschreiben.

3.2 Zustand

While enthält Zuweisungen an (Programm-)Variablen x `:=` a und Zugriff auf Variablen in arithmetischen Ausdrücken. Ein *Zustand* modelliert den Speicher für diese Variablen, der sich während der Ausführung eines Programms von einem Anfangszustand in einen Endzustand entwickelt. Für das formale Modell

ist ein Zustand, hier üblicherweise mit σ bezeichnet, eine Abbildung von Var nach $\mathbb{Z}$, die jeder Variablen x einen Wert $\sigma(x)$ zuordnet. Die Menge aller dieser Zustände sei Σ.

Ein realer Computer muss natürlich viele weitere Informationen im Zustand speichern, die nicht in einem (abstrakten) Zustand aus Σ enthalten sind, z.B. die Zuordnung von Variablen zu Speicheradressen. Diese weiteren Informationen sind aber für die Beschreibung des Verhaltens von While nicht relevant, der abstrakte Zustand und das Modell abstrahieren also davon.

3.3 Semantik von Ausdrücken

Ausdrücke in While liefern einen Wert (Zahl oder Wahrheitswert) zurück, verändern aber den Zustand nicht. Da sie Variablen enthalten können, wird der Wert erst durch einen Zustand endgültig festgelegt.

Für eine Zahl n in Programmdarstellung, in unserem Fall Dezimaldarstellung, liefere $\mathcal{N}[\![n]\!]$ den Wert der Zahl als Element von $\mathbb{Z}$. Beispiel: $\mathcal{N}[\![\texttt{123}]\!] = 123$, wobei $\texttt{123} \in \mathsf{Aexp}$ ein arithmetischer Ausdruck und $123 \in \mathbb{Z}$ eine ganze Zahl ist.

Definition 4 (Semantik arithmetischer Ausdrücke). Für beliebige arithmetische Ausdrücke a definiert $\mathcal{A}[\![a]\!]\sigma$ rekursiv über den Syntaxbaum den Wert von a im Zustand σ:

$$\begin{array}{rcl} \mathcal{A}[\![n]\!]\sigma & = & \mathcal{N}[\![n]\!] \\ \mathcal{A}[\![x]\!]\sigma & = & \sigma(x) \\ \mathcal{A}[\![a_1 \texttt{ - } a_2]\!]\sigma & = & \mathcal{A}[\![a_1]\!]\sigma - \mathcal{A}[\![a_2]\!]\sigma \\ \mathcal{A}[\![a_1 \texttt{ * } a_2]\!]\sigma & = & \mathcal{A}[\![a_1]\!]\sigma \cdot \mathcal{A}[\![a_2]\!]\sigma \end{array}$$

$\mathcal{A}[\![\_]\!]$ ist also eine Funktion des Typs $\mathsf{Aexp} \Rightarrow \Sigma \Rightarrow \mathbb{Z}$, definiert über strukturelle (primitive) Rekursion über den Syntaxbaum arithmetischer Ausdrücke.

Beispiel 5. Was ist $\mathcal{A}[\![a_1 \texttt{ + } a_2]\!]\sigma$?
$a_1 \texttt{ + } a_2$ ist syntaktischer Zucker für $a_1 \texttt{ - (0 - } a_2\texttt{)}$. Damit gilt:

$$\begin{aligned} \mathcal{A}[\![a_1 \texttt{ + } a_2]\!]\sigma = \mathcal{A}[\![a_1 \texttt{ - (0 - } a_2\texttt{)}]\!]\sigma = \mathcal{A}[\![a_1]\!]\sigma - \mathcal{A}[\![\texttt{0 - } a_2]\!]\sigma = \mathcal{A}[\![a_1]\!]\sigma - (\mathcal{A}[\![\texttt{0}]\!]\sigma - \mathcal{A}[\![a_2]\!]\sigma) \\ = \mathcal{A}[\![a_1]\!]\sigma - (0 - \mathcal{A}[\![a_2]\!]\sigma) = \mathcal{A}[\![a_1]\!]\sigma + \mathcal{A}[\![a_2]\!]\sigma \end{aligned}$$

Die syntaktische Abkürzung $a_1 \texttt{ + } a_2$ ist also sinnvoll.

Analog zu arithmetischen Ausdrücken lässt sich der Wert von booleschen Ausdrücken definieren. Dabei bezeichnen **tt** und **ff** die beiden Wahrheitswerte in $\mathbb{B}$.

Definition 6 (Semantik boolescher Ausdrücke).
Die Semantik boolescher Ausdrücke $\mathcal{B}[\![\_]\!]$ ist definiert durch:

$$\begin{array}{rcl} \mathcal{B}[\![\texttt{true}]\!]\sigma & = & \mathbf{tt} \\ \mathcal{B}[\![a_1 \texttt{ <= } a_2]\!]\sigma & = & \mathcal{A}[\![a_1]\!]\sigma \leq \mathcal{A}[\![a_2]\!]\sigma \\ \mathcal{B}[\![\texttt{not } b]\!]\sigma & = & \neg\mathcal{B}[\![b]\!]\sigma \\ \mathcal{B}[\![b_1 \texttt{ \&\& } b_2]\!]\sigma & = & \mathcal{B}[\![b_1]\!]\sigma \wedge \mathcal{B}[\![b_2]\!]\sigma \end{array}$$

Übung: Was ist $\mathcal{B}[\![a_1 \texttt{ == } a_2]\!]\sigma$? Ist die Abkürzung sinnvoll? Was wäre, wenn auch Ausdrücke den Zustand ändern könnten?

4 Operationale Semantik für While

Eine operationale Semantik für While beschreibt nicht nur das Ergebnis einer Programmausführung, sondern auch, wie man zu diesem Ergebnis gelangen kann. Dafür gibt es zwei Modellierungsansätze:

Big-step Semantik (Auswertungssemantik, natural semantics): Hier wird beschrieben, wie man aus dem Verhalten der Teile eines Programmkonstrukts dessen Gesamtverhalten konstruiert.

Small-step Semantik (Transitionssemantik, structural operational semantics): Hier liegt der Fokus auf einzelnen, kleinen Berechnungsschritten, die nach vielen Schritten zum Ende der Programmausführung gelangen.

4.1 Big-Step-Semantik für While

Eine Big-Step Semantik ist eine Auswertungsrelation $\langle c,\ \sigma\rangle \Downarrow \sigma'$, die für ein Programm c und einen Anfangszustand σ bestimmt, ob σ' ein möglicher Endzustand einer Ausführung von c in σ ist.

Definition 7 (Big-Step-Semantik).
Die Auswertungsrelation $\langle c,\ \sigma\rangle \Downarrow \sigma'$ wird durch folgende Regeln induktiv definiert:

$$\textsc{Skip}_{\text{BS}}\colon \langle \texttt{skip},\ \sigma\rangle \Downarrow \sigma \qquad \textsc{Ass}_{\text{BS}}\colon \langle x\ \texttt{:=}\ a,\ \sigma\rangle \Downarrow \sigma[x \mapsto \mathcal{A}[\![a]\!]\,\sigma]$$

$$\textsc{Seq}_{\text{BS}}\colon \frac{\langle c_0,\ \sigma\rangle \Downarrow \sigma' \qquad \langle c_1,\ \sigma'\rangle \Downarrow \sigma''}{\langle c_0\texttt{;}\ c_1,\ \sigma\rangle \Downarrow \sigma''}$$

$$\textsc{IfTT}_{\text{BS}}\colon \frac{\mathcal{B}[\![b]\!]\,\sigma = \mathbf{tt} \qquad \langle c_0,\ \sigma\rangle \Downarrow \sigma'}{\langle \texttt{if (}b\texttt{) then }c_0\texttt{ else }c_1,\ \sigma\rangle \Downarrow \sigma'} \qquad \textsc{IfFF}_{\text{BS}}\colon \frac{\mathcal{B}[\![b]\!]\,\sigma = \mathbf{ff} \qquad \langle c_1,\ \sigma\rangle \Downarrow \sigma'}{\langle \texttt{if (}b\texttt{) then }c_0\texttt{ else }c_1,\ \sigma\rangle \Downarrow \sigma'}$$

$$\textsc{WhileFF}_{\text{BS}}\colon \frac{\mathcal{B}[\![b]\!]\,\sigma = \mathbf{ff}}{\langle \texttt{while (}b\texttt{) do }c,\ \sigma\rangle \Downarrow \sigma}$$

$$\textsc{WhileTT}_{\text{BS}}\colon \frac{\mathcal{B}[\![b]\!]\,\sigma = \mathbf{tt} \qquad \langle c,\ \sigma\rangle \Downarrow \sigma' \qquad \langle \texttt{while (}b\texttt{) do }c,\ \sigma'\rangle \Downarrow \sigma''}{\langle \texttt{while (}b\texttt{) do }c,\ \sigma\rangle \Downarrow \sigma''}$$

Formal ist $\langle \_,\ \_\rangle \Downarrow \_$ die kleinste Menge über $\mathsf{Com} \times \Sigma \times \Sigma$, die unter obigen Regeln abgeschlossen ist. Damit ergibt sich auch folgende Induktionsregel für $\langle \_,\ \_\rangle \Downarrow \_$:

$$\frac{\begin{array}{c}\langle c,\ \sigma\rangle \Downarrow \sigma' \qquad \forall\sigma.\ P(\texttt{skip},\sigma,\sigma) \qquad \forall x,a,\sigma.\ P(x\ \texttt{:=}\ a,\sigma,\sigma[x\mapsto\mathcal{A}[\![a]\!]\,\sigma])\\ \forall c_0,c_1,\sigma,\sigma',\sigma''.\ \langle c_0,\ \sigma\rangle \Downarrow \sigma' \wedge \langle c_1,\ \sigma'\rangle \Downarrow \sigma'' \wedge P(c_0,\sigma,\sigma') \wedge P(c_1,\sigma',\sigma'') \longrightarrow P(c_0\texttt{;}\ c_1,\sigma,\sigma'')\\ \forall b,c_0,c_1,\sigma,\sigma'.\ \mathcal{B}[\![b]\!]\,\sigma = \mathbf{tt} \wedge \langle c_0,\ \sigma\rangle \Downarrow \sigma' \wedge P(c_0,\sigma,\sigma') \longrightarrow P(\texttt{if (}b\texttt{) then }c_0\texttt{ else }c_1,\sigma,\sigma')\\ \forall b,c_0,c_1,\sigma,\sigma'.\ \mathcal{B}[\![b]\!]\,\sigma = \mathbf{ff} \wedge \langle c_1,\ \sigma\rangle \Downarrow \sigma' \wedge P(c_1,\sigma,\sigma') \longrightarrow P(\texttt{if (}b\texttt{) then }c_0\texttt{ else }c_1,\sigma,\sigma')\\ \forall b,c,\sigma.\ \mathcal{B}[\![b]\!]\,\sigma = \mathbf{ff} \longrightarrow P(\texttt{while (}b\texttt{) do }c,\sigma,\sigma)\\ \forall b,c,\sigma,\sigma',\sigma''.\ \mathcal{B}[\![b]\!]\,\sigma = \mathbf{tt} \wedge \langle c,\ \sigma\rangle \Downarrow \sigma' \wedge \langle \texttt{while (}b\texttt{) do }c,\ \sigma'\rangle \Downarrow \sigma'' \wedge\\ P(c,\sigma,\sigma') \wedge P(\texttt{while (}b\texttt{) do }c,\sigma',\sigma'') \longrightarrow P(\texttt{while (}b\texttt{) do }c,\sigma,\sigma'')\end{array}}{P(c,\sigma,\sigma')}$$

Beispiel 8.
Semantik des Programms `z := x; (x := y; y := z)` im Zustand $\sigma_0 = [\mathtt{x}\mapsto 5, \mathtt{y}\mapsto 7, \mathtt{z}\mapsto 0]$ als Ablei-

tungsbaum:

$$\dfrac{\dfrac{}{\langle \texttt{z := x},\ \sigma_0\rangle \Downarrow \sigma_1}\ \textsc{Ass}_{\text{BS}} \qquad \dfrac{\dfrac{}{\langle \texttt{x := y},\ \sigma_1\rangle \Downarrow \sigma_2}\ \textsc{Ass}_{\text{BS}} \qquad \dfrac{}{\langle \texttt{y := z},\ \sigma_2\rangle \Downarrow \sigma_3}\ \textsc{Ass}_{\text{BS}}}{\langle \texttt{x := y; y := z},\ \sigma_1\rangle \Downarrow \sigma_3}\ \textsc{Seq}_{\text{BS}}}{\langle \texttt{z := x; (x := y; y := z)},\ \sigma_0\rangle \Downarrow \sigma_3}\ \textsc{Seq}_{\text{BS}}$$

wobei $\sigma_1 = \sigma_0[\mathtt{z} \mapsto 5]$, $\sigma_2 = \sigma_1[\mathtt{x} \mapsto 7]$ und $\sigma_3 = \sigma_2[\mathtt{y} \mapsto 5]$, also $\sigma_3 = [\mathtt{x} \mapsto 7, \mathtt{y} \mapsto 5, \mathtt{z} \mapsto 5]$.

Übung: Was ist der Ableitungsbaum von folgendem Programm für den Anfangszustand $\sigma_0 = [\mathtt{x} \mapsto 13,\ \mathtt{y} \mapsto 5,\ \mathtt{z} \mapsto 9]$?

```
z := 0; while (y <= x) do (z := z + 1; x := x - y)
```

Lemma 9 (Schleifenabwicklungslemma).
`while` (b) `do` c hat das gleiche Verhalten wie `if` (b) `then` $(c;$ `while` (b) `do` $c)$ `else skip`.

Beweis. Sei $w =$ `while` (b) `do` c und $w' =$ `if` (b) `then` $c;$ `while` (b) `do` c `else skip`. Zu zeigen: $\langle w,\ \sigma\rangle \Downarrow \sigma'$ gilt genau dann, wenn $\langle w',\ \sigma\rangle \Downarrow \sigma'$. Beweis: Fallunterscheidung nach $\mathcal{B}[\![b]\!]\,\sigma$:

- Fall $\mathcal{B}[\![b]\!]\,\sigma = \mathbf{ff}$: Nach den Regeln der Big-Step-Semantik lässt sich $\langle w,\ \sigma\rangle \Downarrow \sigma'$ nur mit der Regel $\textsc{WhileFF}_{\text{BS}}$ ableiten (Regelinversion); $\langle w',\ \sigma\rangle \Downarrow \sigma'$ nur mit den Regeln $\textsc{IfFF}_{\text{BS}}$ und $\textsc{Skip}_{\text{BS}}$. Also:

$$\dfrac{\mathcal{B}[\![b]\!]\,\sigma = \mathbf{ff} \qquad \sigma = \sigma'}{\langle w,\ \sigma\rangle \Downarrow \sigma'} \quad\Leftrightarrow\quad \dfrac{\mathcal{B}[\![b]\!]\,\sigma = \mathbf{ff} \qquad \dfrac{\sigma = \sigma'}{\langle \texttt{skip},\ \sigma\rangle \Downarrow \sigma'}}{\langle w',\ \sigma\rangle \Downarrow \sigma'}$$

- Fall $\mathcal{B}[\![b]\!]\,\sigma = \mathbf{tt}$: Wieder mit Regelinversion gibt es nur die Regel $\textsc{WhileTT}_{\text{BS}}$ für $\langle w,\ \sigma\rangle \Downarrow \sigma'$ und $\textsc{IfTT}_{\text{BS}}$ und $\textsc{Seq}_{\text{BS}}$ für $\langle w',\ \sigma\rangle \Downarrow \sigma'$. Also:

$$\dfrac{\mathcal{B}[\![b]\!]\,\sigma = \mathbf{tt} \qquad \dfrac{A}{\langle c,\ \sigma\rangle \Downarrow \sigma^*} \qquad \dfrac{B}{\langle w,\ \sigma^*\rangle \Downarrow \sigma'}}{\langle w,\ \sigma\rangle \Downarrow \sigma'} \quad\Leftrightarrow\quad \dfrac{\mathcal{B}[\![b]\!]\,\sigma = \mathbf{tt} \qquad \dfrac{\dfrac{A}{\langle c,\ \sigma\rangle \Downarrow \sigma^*} \qquad \dfrac{B}{\langle w,\ \sigma^*\rangle \Downarrow \sigma'}}{\langle c;\ w,\ \sigma\rangle \Downarrow \sigma'}}{\langle w',\ \sigma\rangle \Downarrow \sigma'}$$

□

4.2 Determinismus der Big-Step-Semantik

Eine Semantik ist *deterministisch*, wenn sie jedem Programm und jedem Anfangszustand maximal ein Verhalten zuordnet. Für eine Big-Step-Semantik heißt dies konkret, dass dann auch die Endzustände gleich sind.

Theorem 10 (Determinismus). $\langle \_,\ \_\rangle \Downarrow \_$ ist deterministisch.

Beweis. Zu zeigen: Falls $\langle c,\ \sigma_0\rangle \Downarrow \sigma_1$ und $\langle c,\ \sigma_0\rangle \Downarrow \sigma_2$, dann gilt $\sigma_1 = \sigma_2$.
Beweis: Induktion nach $\langle c,\ \sigma_0\rangle \Downarrow \sigma_1$ (σ_2 beliebig). Damit $P(c, \sigma_0, \sigma_1) \equiv \forall \sigma_2.\ \langle c,\ \sigma_0\rangle \Downarrow \sigma_2 \longrightarrow \sigma_1 = \sigma_2$.

- Fall $\textsc{Skip}_{\text{BS}}$: Zu zeigen: Für alle σ gilt $P(\texttt{skip}, \sigma, \sigma)$, d.h. $\forall \sigma_2.\ \langle \texttt{skip},\ \sigma\rangle \Downarrow \sigma_2 \longrightarrow \sigma = \sigma_2$.
  Sei also σ_2 beliebig mit $\langle \texttt{skip},\ \sigma\rangle \Downarrow \sigma_2$. Aus den Regeln der Big-Step-Semantik lässt sich dies nur mit der Regel $\textsc{Skip}_{\text{BS}}$ ableiten (Regelinversion). Damit folgt $\sigma_2 = \sigma$.
- Fall $\textsc{Ass}_{\text{BS}}$: Zu zeigen: Für alle x, a und σ gilt $P(x\ \texttt{:=}\ a, \sigma, \sigma[x \mapsto \mathcal{A}[\![a]\!]\,\sigma])$, d.h.

$$\forall \sigma_2.\ \langle x\ \texttt{:=}\ a,\ \sigma\rangle \Downarrow \sigma_2 \longrightarrow \sigma[x \mapsto \mathcal{A}[\![a]\!]\,\sigma] = \sigma_2$$

  Sei also σ_2 beliebig mit $\langle x\ \texttt{:=}\ a,\ \sigma\rangle \Downarrow \sigma_2$. Durch Regelinversion ($\textsc{Ass}_{\text{BS}}$) folgt $\sigma_2 = \sigma[x \mapsto \mathcal{A}[\![a]\!]\,\sigma]$.

- Fall SEQ_{BS}: Zu zeigen: Für alle c_0, c_1, σ, σ' und σ'' mit $\langle c_0,\ \sigma\rangle \Downarrow \sigma'$ und $\langle c_1,\ \sigma'\rangle \Downarrow \sigma''$ gilt: Aus $P(c_0, \sigma, \sigma')$ und $P(c_1, \sigma', \sigma'')$ folgt $P(c_0\texttt{;}\ c_1, \sigma, \sigma'')$, d.h.:

$$\begin{aligned}&\big(\forall\sigma_2.\ \langle c_0,\ \sigma\rangle \Downarrow \sigma_2 \longrightarrow \sigma' = \sigma_2\big) \wedge \big(\forall\sigma_2.\ \big\langle c_1,\ \sigma'\big\rangle \Downarrow \sigma_2 \longrightarrow \sigma'' = \sigma_2\big)\\ &\longrightarrow \big(\forall\sigma_2.\ \langle c_0\texttt{;}\ c_1,\ \sigma\rangle \Downarrow \sigma_2 \longrightarrow \sigma'' = \sigma_2\big)\end{aligned}$$

  Sei also σ_2 beliebig mit $\langle c_0\texttt{;}\ c_1,\ \sigma\rangle \Downarrow \sigma_2$. Mit Regelinversion ($\text{SEQ}_{\text{BS}}$) gibt es ein σ^*, so dass $\langle c_0,\ \sigma\rangle \Downarrow \sigma^*$ und $\langle c_1,\ \sigma^*\rangle \Downarrow \sigma_2$. Nach Induktionsannahme $P(c_0, \sigma, \sigma')$ folgt aus $\langle c_0,\ \sigma\rangle \Downarrow \sigma^*$, dass $\sigma' = \sigma^*$. Damit gilt auch $\langle c_1,\ \sigma'\rangle \Downarrow \sigma_2$ und mit der Induktionsannahme $P(c_1, \sigma', \sigma'')$ folgt die Behauptung $\sigma'' = \sigma_2$.

- Fall IFTT_{BS}: Zu zeigen: Für alle b, c_0, c_1, σ und σ' mit $\mathcal{B}[\![b]\!]\,\sigma = \mathbf{tt}$ und $\langle c_0,\ \sigma\rangle \Downarrow \sigma'$ gilt: Aus $P(c_0, \sigma, \sigma')$ folgt $P(\texttt{if (}b\texttt{) then }c_0\texttt{ else }c_1, \sigma, \sigma')$.
  Sei also σ_2 beliebig mit $\langle \texttt{if (}b\texttt{) then }c_0\texttt{ else }c_1,\ \sigma\rangle \Downarrow \sigma_2$, was nur durch die Regeln IFTT_{BS} und IFFF_{BS} ableitbar sein könnte. Wegen $\mathcal{B}[\![b]\!]\,\sigma = \mathbf{tt}$ ist IFFF_{BS} ausgeschlossen. Damit folgt dass $\langle c_0,\ \sigma\rangle \Downarrow \sigma_2$ und mit der Induktionsannahme $P(c_0, \sigma, \sigma')$ die Behauptung $\sigma' = \sigma_2$.

- Fall IFFF_{BS}: Analog zu IFTT_{BS}.

- Fall $\text{WHILETT}_{\text{BS}}$:
  Zu zeigen: Für alle b, c, σ, σ' und σ'' mit $\mathcal{B}[\![b]\!]\,\sigma = \mathbf{tt}$, $\langle c,\ \sigma\rangle \Downarrow \sigma'$ und $\langle \texttt{while (}b\texttt{) do }c,\ \sigma'\rangle \Downarrow \sigma''$ gilt: Aus $P(c, \sigma, \sigma')$ und $P(\texttt{while (}b\texttt{) do }c, \sigma', \sigma'')$ folgt $P(\texttt{while (}b\texttt{) do }c, \sigma, \sigma'')$.
  Sei also σ_2 beliebig mit $\langle \texttt{while (}b\texttt{) do }c,\ \sigma\rangle \Downarrow \sigma_2$. Mit Regelinversion ($\text{WHILETT}_{\text{BS}}$, $\mathcal{B}[\![b]\!]\,\sigma = \mathbf{tt}$ schließt $\text{WHILEFF}_{\text{BS}}$ aus) gibt es ein σ^*, so dass $\langle c,\ \sigma\rangle \Downarrow \sigma^*$ und $\langle \texttt{while (}b\texttt{) do }c,\ \sigma^*\rangle \Downarrow \sigma_2$. Aus $\langle c,\ \sigma\rangle \Downarrow \sigma^*$ folgt mit der Induktionsannahme $P(c, \sigma, \sigma')$, dass $\sigma' = \sigma^*$. Damit folgt mit der Induktionsannahme $P(\texttt{while (}b\texttt{) do }c, \sigma', \sigma'')$ aus $\langle \texttt{while (}b\texttt{) do }c,\ \sigma^*\rangle \Downarrow \sigma_2$ die Behauptung, dass $\sigma'' = \sigma_2$.

- Fall $\text{WHILEFF}_{\text{BS}}$: Zu zeigen: Für alle b, c und σ mit $\mathcal{B}[\![b]\!]\,\sigma = \mathbf{ff}$ gilt $P(\texttt{while (}b\texttt{) do }c, \sigma, \sigma)$.
  Sei also σ_2 beliebig mit $\langle \texttt{while (}b\texttt{) do }c,\ \sigma\rangle \Downarrow \sigma_2$. Nach Regelinversion ($\text{WHILEFF}_{\text{BS}}$, $\mathcal{B}[\![b]\!]\,\sigma = \mathbf{ff}$ schließt $\text{WHILETT}_{\text{BS}}$ aus) folgt die Behauptung $\sigma = \sigma_2$. □

4.3 Small-Step-Semantik für While

Kern einer Small-Step-Semantik ist eine Ein-Schritt-Auswertungsrelation $\langle c,\ \sigma\rangle \to_1 \langle c',\ \sigma'\rangle$, die für ein Programm c und Zustand σ einen einzelnen Rechenschritt beschreibt: c' ist der Rest des Programms, der noch im neuen Zustand σ' auszuführen verbleibt.

Definition 11 (Small-Step-Semantik für While). Die Ein-Schritt-Auswertungsrelation $\to_1$ der Small-Step-Semantik ist induktiv über den Syntaxbaum definiert durch

$$\text{ASS}_{\text{SS}}\colon \langle x\ \texttt{:=}\ a,\ \sigma\rangle \to_1 \langle \texttt{skip},\ \sigma[x \mapsto \mathcal{A}[\![a]\!]\,\sigma]\rangle$$

$$\text{SEQ1}_{\text{SS}}\colon \frac{\langle c_0,\ \sigma\rangle \to_1 \langle c_0',\ \sigma'\rangle}{\langle c_0\texttt{;}\ c_1,\ \sigma\rangle \to_1 \langle c_0'\texttt{;}\ c_1,\ \sigma'\rangle} \qquad \text{SEQ2}_{\text{SS}}\colon \langle \texttt{skip;}\ c,\ \sigma\rangle \to_1 \langle c,\ \sigma\rangle$$

$$\text{IFTT}_{\text{SS}}\colon \frac{\mathcal{B}[\![b]\!]\,\sigma = \mathbf{tt}}{\langle \texttt{if (}b\texttt{) then }c_0\texttt{ else }c_1,\ \sigma\rangle \to_1 \langle c_0,\ \sigma\rangle}$$

$$\text{IFFF}_{\text{SS}}\colon \frac{\mathcal{B}[\![b]\!]\,\sigma = \mathbf{ff}}{\langle \texttt{if (}b\texttt{) then }c_0\texttt{ else }c_1,\ \sigma\rangle \to_1 \langle c_1,\ \sigma\rangle}$$

$$\text{WHILE}_{\text{SS}}\colon \langle \texttt{while (}b\texttt{) do }c,\ \sigma\rangle \to_1 \langle \texttt{if (}b\texttt{) then (}c\texttt{; while (}b\texttt{) do }c\texttt{) else skip},\ \sigma\rangle$$

Definition 12 (blockiert).
Eine Konfiguration, die nicht weiter auswerten kann, ist *blockiert*, notiert als $\langle c,\ \sigma\rangle \not\to_1$.

Für die Anweisung `skip` gibt es keine Auswertungsregel: $\langle \texttt{skip},\ \sigma\rangle$ bezeichnet eine Endkonfiguration des Programms, σ ist der Endzustand. Kennzeichen einer guten Semantik ist, dass von allen (wohlgeformten) Konfigurationen nur Endkonfigurationen blockiert sind.

Definition 13 (Ableitungsfolge). Eine *Ableitungsfolge* für $\gamma_0 = \langle c,\ \sigma\rangle$ ist eine (endliche oder unendliche) Folge $(\gamma_i)_i$ mit $\gamma_0 \to_1 \gamma_1 \to_1 \gamma_2 \to_1 \ldots$. Sie ist *maximal*, falls $(\gamma_i)_i$ unendlich ist oder das letzte γ_k keine Reduktion in $\to_1$ besitzt. $\gamma \overset{n}{\to}_1 \gamma'$ ($\gamma \overset{*}{\to}_1 \gamma'$) bezeichne, dass es eine Ableitungsfolge mit n (endlich vielen) Schritten von γ nach γ' gibt. $\overset{*}{\to}_1$ ist die reflexive, transitive Hülle von $\to_1$.

Maximale Ableitungsfolgen beschreiben das Programmverhalten. Nichtterminierende Ausführungen entsprechen unendlichen Ableitungsfolgen – diese haben *keinen* Endzustand; $\gamma \overset{\infty}{\to}_1$ bezeichne, dass es eine unendlich lange Ableitungsfolge gibt, die in γ beginnt.

Beispiel 14. Semantik des Programms `z := x; (x := y; y := z)` im Zustand $\sigma_0 = [\mathtt{x}\mapsto 5, \mathtt{y}\mapsto 7, \mathtt{z}\mapsto 0]$ als maximale Ableitungsfolge:

$$\begin{aligned}\langle \texttt{z := x; (x := y; y := z)},\ \sigma_0\rangle &\to_1 \langle \texttt{skip; (x := y; y := z)},\ \sigma_1\rangle\\ &\to_1 \langle \texttt{x := y; y := z},\ \sigma_1\rangle\\ &\to_1 \langle \texttt{skip; y := z},\ \sigma_2\rangle\\ &\to_1 \langle \texttt{y := z},\ \sigma_2\rangle\\ &\to_1 \langle \texttt{skip},\ \sigma_3\rangle\end{aligned}$$

wobei $\sigma_1 = \sigma_0[\mathtt{z}\mapsto 5]$, $\sigma_2 = \sigma_1[\mathtt{x}\mapsto 7]$ und $\sigma_3 = \sigma_2[\mathtt{y}\mapsto 5]$, also $\sigma_3 = [\mathtt{x}\mapsto 7, \mathtt{y}\mapsto 5, \mathtt{z}\mapsto 5]$. Jeder einzelne Schritt muss dabei durch einen Ableitungsbaum für $\to_1$ gerechtfertigt werden, z. B.:

$$\dfrac{\dfrac{}{\langle \texttt{z := x},\ \sigma_0\rangle \to_1 \langle \texttt{skip},\ \sigma_1\rangle}\ \textsc{Ass}_{\textsc{ss}}}{\langle \texttt{z := x; (x := y; y := z)},\ \sigma_0\rangle \to_1 \langle \texttt{skip; (x := y; y := z)},\ \sigma_1\rangle}\ \textsc{Seq1}_{\textsc{ss}}$$

Beispiel 15 (Nichttermination).
Sei $w = \texttt{while (not (x == 1)) do x := x + 1}$ und $\sigma_n = [\mathtt{x}\mapsto n]$. Für $\langle w,\ \sigma_0\rangle$ ergibt sich folgende maximale (endliche) Ableitungsfolge:

$$\begin{aligned}\langle w,\ \sigma_0\rangle \to_1 &\langle \overbrace{\texttt{if (not (x == 1)) then x := x + 1; } w \texttt{ else skip}}^{=\text{if}},\ \sigma_0\rangle\\ \to_1 &\langle \texttt{x := x + 1; } w,\ \sigma_0\rangle \to_1 \langle \texttt{skip; } w,\ \sigma_1\rangle \to_1 \langle w,\ \sigma_1\rangle \to_1 \langle \text{if},\ \sigma_1\rangle \to_1 \langle \texttt{skip},\ \sigma_1\rangle\end{aligned}$$

Für $\langle w,\ \sigma_2\rangle$ ist die maximale Ableitungsfolge[1] unendlich:

$$\begin{aligned}&\langle w,\ \sigma_2\rangle \to_1 \langle \text{if},\ \sigma_2\rangle \to_1 \langle \texttt{x := x + 1; } w,\ \sigma_2\rangle \to_1 \langle \texttt{skip; } w,\ \sigma_3\rangle\\ \to_1&\langle w,\ \sigma_3\rangle \to_1 \langle \text{if},\ \sigma_3\rangle \to_1 \langle \texttt{x := x + 1; } w,\ \sigma_3\rangle \to_1 \langle \texttt{skip; } w,\ \sigma_4\rangle\\ \to_1&\langle w,\ \sigma_4\rangle \to_1 \ldots\end{aligned}$$

Häufig beschreiben die einzelnen Schritte maximaler Ableitungsfolgen zu detailliert, was ein Programm berechnet. Beispielsweise haben die Programm `skip; skip` und `skip` unterschiedliche maximale Ableitungsfolgen und damit eine unterschiedliche Semantik. Deswegen abstrahiert man üblicherweise

[1] Für While sind maximale Ableitungsfolgen eindeutig, s. Kor. 18 unten

von den maximalen Ableitungsfolgen und nimmt die transitive Hülle $\overset{*}{\to}_1$ zu einer Endkonfiguration beziehungsweise $\overset{\infty}{\to}_1$ als Semantik eines Programms.

Formal gesehen sind $\overset{*}{\to}_1$ und $\overset{n}{\to}_1$ ganz unterschiedlich definiert. Es gilt aber

$$\langle c,\ \sigma\rangle \overset{*}{\to}_1 \langle c',\ \sigma'\rangle \quad \text{gdw.} \quad \exists n.\ \langle c,\ \sigma\rangle \overset{n}{\to}_1 \langle c',\ \sigma'\rangle$$

Deswegen werden wir im Folgenden bei Bedarf von der $\overset{*}{\to}_1$ auf $\overset{n}{\to}_1$ und wieder zurück wechseln – unter Beachtung, dass n existenziell quantifiziert ist.

Lemma 16 (Fortschritt). `skip` ist das einzige Programm, das blockiert ist.

Beweis. Zu zeigen: Für alle Programme c außer `skip` und jeden Zustand σ gibt es c' und σ' mit $\langle c,\ \sigma\rangle \to_1 \langle c',\ \sigma'\rangle$. Beweis mittels Induktion über c:

- Fall $c =$ `skip`: Explizit ausgeschlossen.
- Fälle $c =$ `x := a`, $c =$ `if (b) then` c_1 `else` c_2 und $c =$ `while (b) do` c_0:
  Folgen direkt aus den Regeln Ass_{SS}, IfTT_{SS}, IfFF_{SS} (Fallunterscheidung nach $\mathcal{B}[\![b]\!]\sigma$) und While_{SS}.
- Fall $c = c_1$`;` c_2: Induktionshypothesen:

  (i) Falls $c_1 \neq$ `skip`, dann gibt es c_1' und σ_1' mit $\langle c_1,\ \sigma\rangle \to_1 \langle c_1',\ \sigma_1'\rangle$.

  (ii) Falls $c_2 \neq$ `skip`, dann gibt es c_2' und σ_2' mit $\langle c_2,\ \sigma\rangle \to_1 \langle c_2',\ \sigma_2'\rangle$.

  Zu zeigen: Es gibt c' und σ' mit $\langle c_1$`;` $c_2,\ \sigma\rangle \to_1 \langle c',\ \sigma'\rangle$.
  Fallunterscheidung nach $c_1 =$ `skip`:
  - Fall $c_1 =$ `skip`: Folgt direkt aus Regel Seq2_{SS}
  - Fall $c_1 \neq$ `skip`: Mit Induktionshypothese (i) gibt es c_1' und σ_1' mit $\langle c_1,\ \sigma\rangle \to_1 \langle c_1',\ \sigma_1'\rangle$. Mit Regel Seq1_{SS} folgt $\langle c_1$`;` $c_2,\ \sigma\rangle \to_1 \langle c_1'$`;` $c_2,\ \sigma_1'\rangle$. □

Im Progress-Beweis wurden alle Regeln für $\to_1$ benutzt, keine ist also überflüssig.

Theorem 17 (Determinismus). $\langle \_,\ \_\rangle \to_1 \langle \_,\ \_\rangle$ ist deterministisch.

Beweis analog zum Determinismus für die Big-Step-Semantik $\langle \_,\ \_\rangle \Downarrow \_$ (Thm. 10).

Korollar 18. Für alle c und σ gibt es genau eine maximale Ableitungsfolge.

Die Existenz einer maximalen Ableitungsfolge folgt aus der Existenz unendlich langer Folgen, die Eindeutigkeit aus dem Determinismus durch Induktion.

4.4 Äquivalenz zwischen Big-Step- und Small-Step-Semantik

Big-Step- und Small-Step-Semantik sind zwei Semantik-Definitionen für While. Bei der Big-Step-Semantik interessiert nur der Endzustand einer Programmausführung, während eine Ableitungsfolge in der Small-Step-Semantik zusätzlich alle einzelnen Zwischenberechnungsschritte und -zustände enthält. Trotzdem passen beide Definitionen wie folgt zusammen, was in diesem Abschnitt bewiesen wird:

$$\langle c,\ \sigma\rangle \Downarrow \sigma' \text{ genau dann, wenn } \langle c,\ \sigma\rangle \overset{*}{\to}_1 \langle \texttt{skip},\ \sigma'\rangle$$

Die Äquivalenzbeweise benötigen die beiden folgenden Lemmas für Sequenz. Gäbe es mehr Sprachkonstrukte mit einer rekursiven Regel für die Small-Step-Semantik wie Seq1_{SS}, bräuchte man entsprechende Lemmas für jedes dieser.

Lemma 19 (Liftinglemma für Sequenz). Falls $\langle c,\ \sigma\rangle \overset{n}{\to}_1 \langle c',\ \sigma'\rangle$, dann $\langle c;\ c_2,\ \sigma\rangle \overset{n}{\to}_1 \langle c';\ c_2,\ \sigma'\rangle$.

Beweis. Induktion über n, der Induktionsschritt folgt aus der Regel $\textsc{Seq1}_{\text{SS}}$. □

Lemma 20 (Zerlegungslemma für Sequenz). Wenn $\langle c_1;\ c_2,\ \sigma\rangle \overset{n}{\to}_1 \langle \texttt{skip},\ \sigma''\rangle$, dann gibt es i, j und σ', so dass $\langle c_1,\ \sigma\rangle \overset{i}{\to}_1 \langle \texttt{skip},\ \sigma'\rangle$ und $\langle c_2,\ \sigma'\rangle \overset{j}{\to}_1 \langle \texttt{skip},\ \sigma''\rangle$ mit $i + j + 1 = n$.

Beweis. Beweis per Induktion über n (c_1 und σ beliebig):

- Basisfall $n = 0$: Dieser Fall ist unmöglich, weil $c_1;\ c_2 \neq \texttt{skip}$.
- Induktionsschritt $n + 1$: Induktionsannahme: Für alle c_1 und σ gilt: Wenn $\langle c_1;\ c_2,\ \sigma\rangle \overset{n}{\to}_1 \langle \texttt{skip},\ \sigma''\rangle$, dann gibt es i, j und σ' mit $\langle c_1,\ \sigma\rangle \overset{i}{\to}_1 \langle \texttt{skip},\ \sigma'\rangle$, $\langle c_2,\ \sigma'\rangle \overset{j}{\to}_1 \langle \texttt{skip},\ \sigma''\rangle$ und $i + j + 1 = n$.

  Unter der Annahme $\langle c_1;\ c_2,\ \sigma\rangle \overset{n+1}{\to}_1 \langle \texttt{skip},\ \sigma''\rangle$ ist zu zeigen, dass es i, j und σ' gibt mit $\langle c_1,\ \sigma\rangle \overset{i}{\to}_1 \langle \texttt{skip},\ \sigma'\rangle$, $\langle c_2,\ \sigma'\rangle \overset{j}{\to}_1 \langle \texttt{skip},\ \sigma''\rangle$ und $i + j + 1 = n + 1$.

  Beweis: Wegen $\langle c_1;\ c_2,\ \sigma\rangle \overset{n+1}{\to}_1 \langle \texttt{skip},\ \sigma''\rangle$ gibt es ein c und σ^*, so dass

  $$\langle c_1;\ c_2,\ \sigma\rangle \to_1 \langle c,\ \sigma^*\rangle \overset{n}{\to}_1 \langle \texttt{skip},\ \sigma''\rangle.$$

  Mit Regelinversion folgt aus $\langle c_1;\ c_2,\ \sigma\rangle \to_1 \langle c,\ \sigma^*\rangle$, dass entweder ($\textsc{Seq2}_{\text{SS}}$) $c_1 = \texttt{skip}$, $c = c_2$, $\sigma^* = \sigma$ oder ($\textsc{Seq1}_{\text{SS}}$) c von der Form $c_1';\ c_2$ mit $\langle c_1,\ \sigma\rangle \to_1 \langle c_1',\ \sigma^*\rangle$ ist. Im ersten Fall folgt die Behauptung mit der Aufteilung $i = 0$, $j = n$ und $\sigma' = \sigma$. Im anderen Fall ergibt die Induktionsannahme für $\langle c_1';\ c_2,\ \sigma^*\rangle \overset{n}{\to}_1 \langle \texttt{skip},\ \sigma''\rangle$ eine Aufteilung in $\langle c_1',\ \sigma^*\rangle \overset{i'}{\to}_1 \langle \texttt{skip},\ \sigma'\rangle$ und $\langle c_2,\ \sigma'\rangle \overset{j'}{\to}_1 \langle \texttt{skip},\ \sigma''\rangle$ mit $i' + j' + 1 = n$. Mit $\langle c_1,\ \sigma\rangle \to_1 \langle c_1',\ \sigma^*\rangle$ ergibt sich dann die Behauptung für $i = i' + 1$, $j = j'$ und $\sigma' = \sigma'$. □

Theorem 21 (Small-Step simuliert Big-Step). Aus $\langle c,\ \sigma\rangle \Downarrow \sigma'$ folgt $\langle c,\ \sigma\rangle \overset{*}{\to}_1 \langle \texttt{skip},\ \sigma'\rangle$.

Beweis in der Übung.

Theorem 22 (Big-Step simuliert Small-Step). Aus $\langle c,\ \sigma\rangle \overset{*}{\to}_1 \langle \texttt{skip},\ \sigma'\rangle$ folgt $\langle c,\ \sigma\rangle \Downarrow \sigma'$.

Beweis. Wegen $\langle c,\ \sigma\rangle \overset{*}{\to}_1 \langle \texttt{skip},\ \sigma'\rangle$ gibt es ein n mit $\langle c,\ \sigma\rangle \overset{n}{\to}_1 \langle \texttt{skip},\ \sigma'\rangle$. Beweis von $\langle c,\ \sigma\rangle \Downarrow \sigma'$ per *vollständiger Induktion* über n (c, σ, σ' beliebig):

Sei n beliebig. Induktionsannahme: Für alle $m < n$ und c, σ, σ' gilt: Wenn $\langle c,\ \sigma\rangle \overset{m}{\to}_1 \langle \texttt{skip},\ \sigma'\rangle$, dann auch $\langle c,\ \sigma\rangle \Downarrow \sigma'$. Zu zeigen: Aus (i) $\langle c,\ \sigma\rangle \overset{n}{\to}_1 \langle \texttt{skip},\ \sigma'\rangle$ folgt $\langle c,\ \sigma\rangle \Downarrow \sigma'$ für beliebige c, σ und σ'.

Fallunterscheidung nach c:

- Fall $c = \texttt{skip}$: Mit (i) folgt, dass $n = 0$ und $\sigma' = \sigma$. $\langle \texttt{skip},\ \sigma\rangle \Downarrow \sigma$ folgt aus Regel $\textsc{Skip}_{\text{BS}}$.
- Fall $c = \texttt{x := } a$: Mit (i) folgt, dass $n = 1$ und $\sigma' = \sigma[\texttt{x} \mapsto \mathcal{A}[\![a]\!]\sigma]$. $\langle \texttt{x := } a,\ \sigma\rangle \Downarrow \sigma[\texttt{x} \mapsto \mathcal{A}[\![a]\!]\sigma]$ folgt aus der Regel Ass_{BS}.
- Fall $c = c_1;\ c_2$:

  Nach dem Zerlegungslemma lässt sich (i) in $\langle c_1,\ \sigma\rangle \overset{i}{\to}_1 \langle \texttt{skip},\ \sigma^*\rangle$ und $\langle c_2,\ \sigma^*\rangle \overset{j}{\to}_1 \langle \texttt{skip},\ \sigma'\rangle$ mit $i + j + 1 = n$ aufteilen. Damit ist insbesondere $i < n$ und $j < n$, d.h., die Induktionsannahme lässt sich auf beide Teile anwenden: $\langle c_1,\ \sigma\rangle \Downarrow \sigma^*$ und $\langle c_2,\ \sigma^*\rangle \Downarrow \sigma'$. Daraus folgt die Behauptung mit der Regel $\textsc{Seq}_{\text{BS}}$.
- Fall $c = \texttt{if (}b\texttt{) then } c_1 \texttt{ else } c_2$: Aus (i) folgt mit Regelinversion, dass $n > 0$ und entweder ($\textsc{IfTT}_{\text{SS}}$) $\mathcal{B}[\![b]\!]\sigma = \mathbf{tt}$ und $\langle c_1,\ \sigma\rangle \overset{n-1}{\to}_1 \langle \texttt{skip},\ \sigma'\rangle$ oder ($\textsc{IfFF}_{\text{SS}}$) $\mathcal{B}[\![b]\!]\sigma = \mathbf{ff}$ und $\langle c_2,\ \sigma\rangle \overset{n-1}{\to}_1 \langle \texttt{skip},\ \sigma'\rangle$. In beiden Fällen lässt sich die Induktionsannahme anwenden und die Behauptung folgt aus den Regeln $\textsc{IfTT}_{\text{BS}}$ bzw. $\textsc{IfFF}_{\text{BS}}$.

- Fall $c = \texttt{while (}b\texttt{) do } c$: Aus (i) folgt mit Regelinversion (WHILE_{SS}), dass $n > 0$ und

$$\langle \texttt{while (}b\texttt{) do } c,\ \sigma\rangle \to_1 \underbrace{\langle \texttt{if (}b\texttt{) then } c\texttt{; while (}b\texttt{) do } c \texttt{ else skip},\ \sigma\rangle}_{=w'} \overset{n-1}{\to}_1 \langle \texttt{skip},\ \sigma'\rangle.$$

  Wendet man die Induktionshypothese mit $m = n - 1 < n$ und $c = w'$ an, so folgt $\langle w',\ \sigma\rangle \Downarrow \sigma'$. Da w' nach dem Schleifenabwicklungslemma (Lem. 9) in der Big-Step-Semantik äquivalent zu $\texttt{while (}b\texttt{) do } c$ ist, gilt auch $\langle \texttt{while (}b\texttt{) do } c,\ \sigma\rangle \Downarrow \sigma'$. □

Korollar 23 (Äquivalenz von Big-Step- und Small-Step-Semantik).
Für alle c, σ und σ' gilt $\langle c,\ \sigma\rangle \Downarrow \sigma'$ genau dann, wenn $\langle c,\ \sigma\rangle \overset{*}{\to}_1 \langle \texttt{skip},\ \sigma'\rangle$ gilt.

4.5 Äquivalenz von Programmen

Zu entscheiden. ob zwei Programme äquivalent sind, ist ein wesentlicher Anwendungsbereich für Semantiken. Die bisherigen Äquivalenzbegriffe sind dafür aber i. d. R. zu feingranular, wie folgendes Beispiel zeigt:

```
tmp := y; y := x; x := tmp        x := x - y; y := y + x; x := y - x
```

Beide Programme vertauschen die Inhalte von `x` und `y`, aber das erste verwendet dazu die Hilfsvariable `tmp`. Demnach sind beide Programme nicht äquivalent, weil das eine `tmp` möglicherweise verändert, das andere aber nicht. Wird im weiteren Programmverlauf der in `tmp` gespeicherte Wert aber nicht mehr verwendet, wäre es gerechtfertigt, beide Programme als äquivalent zu betrachten.

Definition 24 (Äquivalenz von Programmen). Zwei Programme c_1 und c_2 sind äquivalent bezüglich der Variablen $V \subseteq \mathsf{Var}$, falls für alle σ gilt:

- Wenn $\langle c_1,\ \sigma\rangle \Downarrow \sigma_1$ für ein σ_1, dann gibt es ein σ_2 mit $\langle c_2,\ \sigma\rangle \Downarrow \sigma_2$ und $\sigma_1(x) = \sigma_2(x)$ für alle $x \in V$.
- Wenn $\langle c_2,\ \sigma\rangle \Downarrow \sigma_2$ für ein σ_2, dann gibt es ein σ_1 mit $\langle c_1,\ \sigma\rangle \Downarrow \sigma_1$ und $\sigma_1(x) = \sigma_2(x)$ für alle $x \in V$.

In obigem Beispiel sind beide Programme äquivalent bezüglich der Variablen $\{\,\texttt{x}, \texttt{y}\,\}$.

Die beiden Bedingungen sind klassische Simulationsbedingungen in beide Richtungen, man kann sie auch für Small-Step-Semantiken entsprechend formulieren. Da die Big-Step-Semantik deterministisch ist, lassen sie sich wie folgt vereinfachen.

Lemma 25 (Äquivalenz für deterministische Programme). Zwei Programme c_1 und c_2 sind äquivalent bezüglich V genau dann, wenn

(i) c_1 terminiert genau dann, wenn c_2 terminiert, d.h., es gibt ein σ_1 mit $\langle c_1,\ \sigma\rangle \Downarrow \sigma_1$ genau dann, wenn es ein σ_2 mit $\langle c_2,\ \sigma\rangle \Downarrow \sigma_2$ gibt.

(ii) Wenn $\langle c_1,\ \sigma\rangle \Downarrow \sigma_1$ und $\langle c_2,\ \sigma\rangle \Downarrow \sigma_2$, dann $\sigma_1(x) = \sigma_2(x)$ für alle $x \in V$.

5 Ein Compiler für While

Reale Rechner verarbeiten Assembler-Code und keine Syntaxbäume. Sprachen wie While sind damit nicht direkt auf einem solchen Rechner ausführbar, sondern müssen übersetzt werden. Die Regeln der Big-Step-Semantik (und auch der Small-Step-Semantik) lassen sich beispielsweise direkt in Prolog-Regeln konvertieren, die ein Prolog-Interpreter ausführen kann. Der für die Regeln spezialisierte Interpreter führt dann das Programm aus, übersetzt es also in eine Ausführung des Programms auf einem konkreten Rechner. Dabei wird aber das auszuführende Programm selbst nicht in eine für den Rechner geeignetere Darstellung übersetzt.

Direkter geht es, wenn man einen solchen Rechner und seine Instruktionen selbst formal modelliert und einen Übersetzer (Compiler) für While-Programme schreibt, der semantisch äquivalente Programme erzeugt. In diesem Abschnitt wird dieser Ansatz für ein sehr abstraktes Modell eines solchen Rechners für die Sprache While ausgearbeitet.

5.1 Ein abstraktes Modell eines Rechners ASM

Der abstrakte Rechner hat einen Speicher für Daten und eine Liste von Befehlen, die er abarbeitet. In unserem einfachen Modell reichen drei Assembler-Befehle (zusammengefasst in der Menge Asm), zwei zur Kontrollflusssteuerung und eine Datenoperation.

Definition 26 (Instruktionen in ASM).

`ASSN` x Aexp	Zuweisung
`JMP` k	relativer Sprung ($k \in \mathbb{Z}$)
`JMPF` k Bexp	bedingter, relativer Sprung ($k \in \mathbb{Z}$)

Ein Assembler-Programm (ASM) P besteht aus einer unveränderlichen Liste der abzuarbeitenden Befehle, angefangen mit dem ersten der Liste.

Neben dem Zustand für den Variableninhalt gibt ein *Programmzähler* an, die wievielte Instruktion der Liste die nächste ist, die abgearbeitet werden muss. Notation für einen einzelnen Ausführungsschritt: $P \vdash \langle i,\ \sigma\rangle \to \langle i',\ \sigma'\rangle$. Für ein gegebenes Programm (Instruktionsliste) P transformiert die i-te Instruktion in P den Zustand σ in den Zustand σ' und i' bezeichnet die nächste auszuführende Instruktion. P_i bezeichne das i-te Element von P und ist nur definiert, wenn i nicht negativ und kleiner als die Länge von P ist.

Definition 27 (Semantik von ASM).
Die Semantik $P \vdash \langle \_,\ \_\rangle \to \langle \_,\ \_\rangle$ eines ASM-Programms P ist durch folgende Regeln definiert.

$$\textsc{Assn}: \frac{P_i = \texttt{ASSN}\ x\ a}{P \vdash \langle i,\ \sigma\rangle \to \langle i+1,\ \sigma[x \mapsto \mathcal{A}\,[\![a]\!]\,\sigma]\rangle} \qquad \textsc{Jmp}: \frac{P_i = \texttt{JMP}\ k}{P \vdash \langle i,\ \sigma\rangle \to \langle i+k,\ \sigma\rangle}$$

$$\textsc{JmpFT}: \frac{P_i = \texttt{JMPF}\ k\ b \qquad \mathcal{B}\,[\![b]\!]\,\sigma = \mathbf{tt}}{P \vdash \langle i,\ \sigma\rangle \to \langle i+1,\ \sigma\rangle} \qquad \textsc{JmpFF}: \frac{P_i = \texttt{JMPF}\ k\ b \qquad \mathcal{B}\,[\![b]\!]\,\sigma = \mathbf{ff}}{P \vdash \langle i,\ \sigma\rangle \to \langle i+k,\ \sigma\rangle}$$

Wenn i negativ, größer als die, oder gleich der Länge von P ist, ist keine Ausführung möglich.

Die gesamte Semantik eines Programms P in einem Zustand σ ist wieder über die maximalen Ableitungssequenzen von $\langle 0,\ \sigma\rangle$ gegeben; oder aber durch die blockierten Konfigurationen $\langle i',\ \sigma\rangle$, die in der transitiven Hülle $P \vdash \langle i,\ \sigma\rangle \xrightarrow{*} \langle i',\ \sigma'\rangle$ von $\langle 0,\ \sigma\rangle$ aus erreichbar sind, und die Existenz unendlicher Ausführungen $P \vdash \langle 0,\ \sigma\rangle \xrightarrow{\infty}$.

Übung: Welche Konstrukte und Regeln der Assembler-Sprache sind überflüssig und könnten durch die anderen simuliert werden?

5.2 Ein Compiler von While nach ASM

Sei $P +\!\!+ P'$ die Verkettung der beiden Listen P und P' und $|P|$ die Länge von P.

Definition 28 (Compiler). Der Compiler von While nach ASM sei durch die Funktion comp definiert:

$$\begin{aligned}
\text{comp}(\texttt{skip}) &= [] \\
\text{comp}(x\ \texttt{:=}\ a) &= [\texttt{ASSN}\ x\ a] \\
\text{comp}(c_1\texttt{;}\ c_2) &= \text{comp}(c_1) +\!\!+ \text{comp}(c_2) \\
\text{comp}(\texttt{if (}b\texttt{) then}\ c_1\ \texttt{else}\ c_2) &= [\texttt{JMPF}\ k_1\ b] +\!\!+ \text{comp}(c_1) +\!\!+ [\texttt{JMP}\ k_2] +\!\!+ \text{comp}(c_2) \\
&\quad \text{wobei } k_1 = |\text{comp}(c_1)| + 2 \text{ und } k_2 = |\text{comp}(c_2)| + 1 \\
\text{comp}(\texttt{while (}b\texttt{) do}\ c) &= [\texttt{JMPF}\ (k+2)\ b] +\!\!+ \text{comp}(c) +\!\!+ [\texttt{JMP}\ -(k+1)] \\
&\quad \text{wobei } k = |\text{comp}(c)|
\end{aligned}$$

Beispiel 29.
Das Kompilat des Programms `z := 0; while (y <= x) do (z := z + 1; x := x - y)` ist:

$$[\texttt{ASSN z 0},\ \texttt{JMPF}\ 4\ (\texttt{y <= x}),\ \texttt{ASSN z (z + 1)},\ \texttt{ASSN x (x - y)},\ \texttt{JMP}\ -3]$$

Für `(if (x <= y) then x := x + y; y := x - y; x := x - y else y := x); z := 5` ergibt sich folgendes Kompilat – unabhängig von der Klammerung der Sequenz im `then`-Zweig:

$$[\texttt{JMPF}\ 5\ (\texttt{x <= y}),\ \texttt{ASSN x (x + y)},\ \texttt{ASSN y (x - y)},\ \texttt{ASSN x (x - y)},\ \texttt{JMP}\ 2,\ \texttt{ASSN y x},\ \texttt{ASSN z 5}\ ]$$

Übersetzt man `if (x <= -1) then x := -1 * x else skip`, so erhält man:

$$[\texttt{JMPF}\ 3\ (\texttt{x <= -1}),\ \texttt{ASSN x (-1 * x)},\ \texttt{JMP}\ 1]$$

5.3 Korrektheit des Compilers

Ein Compiler soll die Semantik eines Programms nicht verändern. Dadurch, dass die Semantik von While und ASM formal gegeben sind, lässt sich das auch exakt formulieren und beweisen:

- Wenn $\langle c,\ \sigma\rangle \Downarrow \sigma'$, dann $\text{comp}(c) \vdash \langle 0,\ \sigma\rangle \xrightarrow{*} \langle |\text{comp}(c)|,\ \sigma'\rangle$.
- Wenn es keine Big-Step-Ableitung für $\langle c,\ \sigma\rangle$ gibt, dann $\text{comp}(c) \vdash \langle 0,\ \sigma\rangle \xrightarrow{\infty}$.

Theorem 30 (ASM simuliert Big-Step).
Wenn $\langle c,\ \sigma\rangle \Downarrow \sigma'$, dann $\text{comp}(c) \vdash \langle 0,\ \sigma\rangle \xrightarrow{*} \langle |\text{comp}(c)|,\ \sigma'\rangle$.

Dieses Theorem folgt direkt aus folgender Verallgemeinerung, die erlaubt, dass die Maschinenbefehlssequenz in beliebigen Code eingebettet ist.

Lemma 31. Seien P_1 und P_2 beliebige ASM Programme.
Wenn $\langle c,\ \sigma\rangle \Downarrow \sigma'$, dann $P_1 +\!+ \text{comp}(c) +\!+ P_2 \vdash \langle |P_1|,\ \sigma\rangle \xrightarrow{*} \langle |P_1| + |\text{comp}(c)|,\ \sigma'\rangle$.

Beweis. Beweis durch Regelinduktion über $\langle c,\ \sigma\rangle \Downarrow \sigma'$, P_1 und P_2 beliebig. Notation: $|c| = |\text{comp}(c)|$

- Fall SKIP_{BS}: Zu zeigen:
  Für alle P_1 und P_2 gilt $P_1 +\!+ \text{comp}(\texttt{skip}) +\!+ P_2 \vdash \langle |P_1|,\ \sigma\rangle \xrightarrow{*} \langle |P_1| + |\texttt{skip}|,\ \sigma\rangle$.

  Trivial wegen $|\texttt{skip}| = 0$.

- Fall ASS_{BS}: Zu zeigen: Für alle P_1 und P_2 gilt
  $P_1 +\!+ \text{comp}(x\ \texttt{:=}\ a) +\!+ P_2 \vdash \langle |P_1|,\ \sigma\rangle \xrightarrow{*} \langle |P_1| + |x\ \texttt{:=}\ a|,\ \sigma[x \mapsto \mathcal{A}[\![a]\!]\,\sigma]\rangle$.

  Beweis: $P_1 +\!+ [\texttt{ASSN}\ x\ a] +\!+ P_2 \vdash \langle |P_1|,\ \sigma\rangle \to \langle |P_1| + 1,\ \sigma[x \mapsto \mathcal{A}[\![a]\!]\,\sigma]\rangle$ nach Regel ASSN, da $(P_1 +\!+ [\texttt{ASSN}\ x\ a] +\!+ P_2)_{|P_1|} = \texttt{ASSN}\ x\ a$.

- Fall SEQ_{BS}: Zu zeigen: Für alle P_1 und P_2 gilt
  $P_1 +\!+ \text{comp}(c_1\texttt{;}\ c_2) +\!+ P_2 \vdash \langle |P_1|,\ \sigma\rangle \xrightarrow{*} \langle |P_1| + |c_1\texttt{;}\ c_2|,\ \sigma''\rangle$.

  Induktionsannahmen: Für beliebige P_1 und P_2 gelten
  $P_1 +\!+ \text{comp}(c_1) +\!+ P_2 \vdash \langle |P_1|,\ \sigma\rangle \xrightarrow{*} \langle |P_1| + |c_1|,\ \sigma'\rangle$ und
  $P_1 +\!+ \text{comp}(c_2) +\!+ P_2 \vdash \langle |P_1|,\ \sigma'\rangle \xrightarrow{*} \langle |P_1| + |c_2|,\ \sigma''\rangle$.

  Instanziiert man in der ersten Induktionsannahme P_2 mit $\text{comp}(c_2) +\!+ P_2$ und P_1 der zweiten Induktionsannahme mit $P_1 +\!+ \text{comp}(c_1)$, so gelten:

  $$P_1 +\!+ \text{comp}(c_1) +\!+ (\text{comp}(c_2) +\!+ P_2) \vdash \langle |P_1|,\ \sigma\rangle \xrightarrow{*} \langle |P_1| + |c_1|,\ \sigma'\rangle$$
  $$(P_1 +\!+ \text{comp}(c_1)) +\!+ \text{comp}(c_2) +\!+ P_2 \vdash \langle |P_1 +\!+ \text{comp}(c_1)|,\ \sigma'\rangle \xrightarrow{*} \langle |P_1 +\!+ \text{comp}(c_1)| + |c_2|,\ \sigma''\rangle$$

  Ausrechnen und Transitivität von $\xrightarrow{*}$ liefern die Behauptung.

- Fall IFTT_{BS}: Zu zeigen: Für alle P_1 und P_2 gilt $P_1 +\!+ \text{comp}(\texttt{if (}b\texttt{) then}\ c_1\ \texttt{else}\ c_2) +\!+ P_2 \vdash \langle |P_1|,\ \sigma\rangle \xrightarrow{*} \langle |P_1| + |\texttt{if (}b\texttt{) then}\ c_1\ \texttt{else}\ c_2|,\ \sigma'\rangle$.

  Induktionsannahmen:
  $\mathcal{B}[\![b]\!]\,\sigma = \mathbf{tt}$ und für beliebige P_1 und P_2 gilt $P_1 +\!+ \text{comp}(c_1) +\!+ P_2 \vdash \langle |P_1|,\ \sigma\rangle \xrightarrow{*} \langle |P_1| + |c_1|,\ \sigma'\rangle$.

  Beweis mit der Induktionsannahme, bei der P_1 als $P_1 +\!+ [\texttt{JMPF}\ (|c_1| + 2)\ b]$ und
  P_2 als $[\texttt{JMP}\ (|c_2| + 1)] +\!+ \text{comp}(c_2) +\!+ P_2$ instanziiert werden:

  $$P_1 +\!+ [\texttt{JMPF}\ (|c_1| + 2)\ b] +\!+ \text{comp}(c_1) +\!+ [\texttt{JMP}\ (|c_2| + 1)] +\!+ \text{comp}(c_2) +\!+ P_2 \vdash$$
  $$\langle |P_1|,\ \sigma\rangle \to \langle |P_1| + 1,\ \sigma\rangle \xrightarrow{*} \langle |P_1| + 1 + |c_1|,\ \sigma'\rangle \to \langle |P_1| + 2 + |c_1| + |c_2|,\ \sigma'\rangle$$

- Fall IFFF_{BS}: Analog zu IFTT_{BS}.

- Fall $\text{WHILETT}_{\text{BS}}$: Zu zeigen: Für alle P_1 und P_2 gilt
  $P_1 +\!+ \text{comp}(\texttt{while (}b\texttt{) do}\ c) +\!+ P_2 \vdash \langle |P_1|,\ \sigma\rangle \xrightarrow{*} \langle |P_1| + |\texttt{while (}b\texttt{) do}\ c|,\ \sigma''\rangle$.

  Induktionsannahmen: $\mathcal{B}[\![b]\!]\,\sigma = \mathbf{tt}$ und für beliebige P_1 und P_2 gelten
  $P_1 +\!+ \text{comp}(c) +\!+ P_2 \vdash \langle |P_1|,\ \sigma\rangle \xrightarrow{*} \langle |P_1| + |c|,\ \sigma'\rangle$ und $P_1 +\!+ \text{comp}(\texttt{while (}b\texttt{) do}\ c) +\!+ P_2 \vdash \langle |P_1|,\ \sigma'\rangle \xrightarrow{*} \langle |P_1| + |\texttt{while (}b\texttt{) do}\ c|,\ \sigma''\rangle$.

  Beweis mit entsprechend instanziierten Induktionshypothesen und Regel JMPFT:

  $$P_1 +\!+ [\texttt{JMPF}\ (|c| + 2)\ b] +\!+ \text{comp}(c) +\!+ [\texttt{JMP}\ -(|c| + 1)] +\!+ P_2 \vdash$$
  $$\langle |P_1|,\ \sigma\rangle \to \langle |P_1| + 1,\ \sigma\rangle \xrightarrow{*} \langle |P_1| + 1 + |c|,\ \sigma'\rangle \to \langle |P_1|,\ \sigma'\rangle \xrightarrow{*} \langle |P_1| + |\texttt{while (}b\texttt{) do}\ c|,\ \sigma''\rangle$$

- Fall $\text{WHILEFF}_{\text{BS}}$: Induktionsannahme: $\mathcal{B}[\![b]\!]\sigma = \mathbf{ff}$.
  Zu zeigen: Für alle P_1 und P_2 gilt
  $P_1 ++ \text{comp}(\texttt{while (}b\texttt{) do }c) ++ P_2 \vdash \langle |P_1|,\ \sigma\rangle \xrightarrow{*} \langle |P_1| + |\texttt{while (}b\texttt{) do }c|,\ \sigma\rangle$.
  Beweis mit Regel JMPFF:

$$P_1 ++ [\texttt{JMPF } b\ (|c|+2)] ++ \text{comp}(c) ++ [\texttt{JMP } -(|c|+1)] ++ P_2 \ \vdash \langle |P_1|,\ \sigma\rangle \to \langle |P_1| + |c| + 2,\ \sigma\rangle$$

□

Alternativ lässt sich Theorem 30 auch direkt per Induktion beweisen, braucht dann aber folgendes Hilfslemma, das es erlaubt, Instruktionslisten vorne oder hinten zu erweitern:

Lemma 32 (Verschiebungslemma).

(i) Wenn $P \vdash \langle i,\ \sigma\rangle \to \langle i',\ \sigma'\rangle$, dann gilt auch, dass $P' ++ P \vdash \langle i + |P'|,\ \sigma\rangle \to \langle i' + |P'|,\ \sigma'\rangle$ und $P ++ P' \vdash \langle i,\ \sigma\rangle \to \langle i',\ \sigma'\rangle$.

(ii) Wenn $P \vdash \langle i,\ \sigma\rangle \xrightarrow{n} \langle i',\ \sigma'\rangle$, dann gilt auch, dass $P' ++ P \vdash \langle i + |P'|,\ \sigma\rangle \xrightarrow{n} \langle i' + |P'|,\ \sigma'\rangle$ und $P ++ P' \vdash \langle i,\ \sigma\rangle \xrightarrow{n} \langle i',\ \sigma'\rangle$.

Beweis. (i) Fallunterscheidung nach P_i. (ii) Induktion über n, Induktionsschritt mit (i). □

Direkter Beweis von Theorem 30: Beweis durch Regelinduktion über $\langle c,\ \sigma\rangle \Downarrow \sigma'$. Notation $|c| = |\text{comp}(c)|$.

- Fall SKIP_{BS}: Zu zeigen: $\text{comp}(\texttt{skip}) \vdash \langle 0,\ \sigma\rangle \xrightarrow{*} \langle |\texttt{skip}|,\ \sigma\rangle$. Trivial wegen $|\texttt{skip}| = 0$.
- Fall ASS_{BS}: Zu zeigen: $\text{comp}(x\ \texttt{:=}\ a) \vdash \langle 0,\ \sigma\rangle \xrightarrow{*} \langle |x\ \texttt{:=}\ a|,\ \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma]\rangle$.
  Beweis: $[\texttt{ASSN } x\ a] \vdash \langle 0,\ \sigma\rangle \to \langle 1,\ \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma]\rangle$ nach Regel ASSN.
- Fall SEQ_{BS}: Zu zeigen: $\text{comp}(c_1\texttt{;}\ c_2) \vdash \langle 0,\ \sigma\rangle \xrightarrow{*} \langle |c_1\texttt{;}\ c_2|,\ \sigma''\rangle$.
  Induktionsannahmen: $\text{comp}(c_1) \vdash \langle 0,\ \sigma\rangle \xrightarrow{*} \langle |c_1|,\ \sigma'\rangle$ und $\text{comp}(c_2) \vdash \langle 0,\ \sigma'\rangle \xrightarrow{*} \langle |c_2|,\ \sigma''\rangle$.

  Mit dem Verschiebungslemma 32 folgt aus den Induktionsannahmen, dass

$$\begin{aligned}&\text{comp}(c_1) ++ \text{comp}(c_2) \vdash \langle 0,\ \sigma\rangle \xrightarrow{*} \langle |c_1|,\ \sigma'\rangle\\ &\text{comp}(c_1) ++ \text{comp}(c_2) \vdash \langle 0 + |c_1|,\ \sigma'\rangle \xrightarrow{*} \langle |c_2| + |c_1|,\ \sigma''\rangle\end{aligned}$$

  Transitivität von $\xrightarrow{*}$ liefert die Behauptung.
- Fall IFTT_{BS}: Induktionsannahmen: $\text{comp}(c_1) \vdash \langle 0,\ \sigma\rangle \xrightarrow{*} \langle |c_1|,\ \sigma'\rangle$ und $\mathcal{B}[\![b]\!]\sigma = \mathbf{tt}$.
  Zu zeigen: $\text{comp}(\texttt{if (}b\texttt{) then }c_1\texttt{ else }c_2) \vdash \langle 0,\ \sigma\rangle \xrightarrow{*} \langle |\texttt{if (}b\texttt{) then }c_1\texttt{ else }c_2|,\ \sigma'\rangle$.
  Beweis mit dem Verschiebungslemma 32:

$$\begin{aligned}&[\texttt{JMPF } (|c_1|+2)\ b] ++ \text{comp}(c_1) ++ [\texttt{JMP } (|c_2|+1)] ++ \text{comp}(c_2) \ \vdash\\ &\quad \langle 0,\ \sigma\rangle \to \langle 1,\ \sigma\rangle \xrightarrow{*} \langle 1 + |c_1|,\ \sigma'\rangle \to \langle 2 + |c_1| + |c_2|,\ \sigma'\rangle\end{aligned}$$

- Fall IFFF_{BS}: Analog zu IFTT_{BS}.
- Fall $\text{WHILETT}_{\text{BS}}$: Induktionsannahmen: $\mathcal{B}[\![b]\!]\sigma = \mathbf{tt}$, $\text{comp}(c) \vdash \langle 0,\ \sigma\rangle \xrightarrow{*} \langle |c|,\ \sigma'\rangle$ und $\text{comp}(\texttt{while (}b\texttt{) do }c) \vdash \langle 0,\ \sigma'\rangle \xrightarrow{*} \langle |\texttt{while (}b\texttt{) do }c|,\ \sigma''\rangle$.
  Zu zeigen: $\text{comp}(\texttt{while (}b\texttt{) do }c) \vdash \langle 0,\ \sigma\rangle \xrightarrow{*} \langle |\texttt{while (}b\texttt{) do }c|,\ \sigma''\rangle$.
  Beweis mit dem Verschiebungslemma 32:

$$\begin{aligned}&[\texttt{JMPF } (|c|+2)\ b] ++ \text{comp}(c) ++ [\texttt{JMP } -(|c|+1)] \ \vdash\\ &\quad \langle 0,\ \sigma\rangle \to \langle 1,\ \sigma\rangle \xrightarrow{*} \langle |c| + 1,\ \sigma'\rangle \to \langle 0,\ \sigma'\rangle \xrightarrow{*} \langle |\texttt{while (}b\texttt{) do }c|,\ \sigma''\rangle\end{aligned}$$

- Fall $\text{WHILEFF}_{\text{BS}}$: Induktionsannahme: $\mathcal{B}[\![b]\!]\,\sigma = \mathbf{ff}$.
  Zu zeigen: $\text{comp}(\texttt{while (}b\texttt{) do }c) \vdash \langle 0,\ \sigma\rangle \xrightarrow{*} \langle |\texttt{while (}b\texttt{) do }c|,\ \sigma\rangle$.
  Beweis mit Regel JMPFF:

$$[\texttt{JMPF}\ b\ (|c|+2)] +\!\!+ \text{comp}(c) +\!\!+ [\texttt{JMP}\ -(|c|+1)] \vdash \langle 0,\ \sigma\rangle \to \langle |c|+2,\ \sigma\rangle \qquad \square$$

Dieses Theorem zeigt, dass es zu jeder terminierenden Ausführung in der Big-Step-Semantik eine entsprechende (terminierende) Ausführung des übersetzten Programms gibt. Das Theorem sagt jedoch nichts über nicht terminierende Programme aus: Wir haben den zweiten Teil des obigen Korrektheitsbegriffs – nicht terminierende Ausführungen in While terminieren auch in ASM nicht – noch nicht bewiesen. Da Nichttermination aber in der Big-Step-Semantik nicht direkt ausdrückbar ist, wäre ein direkter Beweis formal sehr aufwändig. Stattdessen zeigt man üblicherweise die umgekehrte Simulationsrichtung: Wenn $\text{comp}(c) \vdash \langle 0,\ \sigma\rangle \xrightarrow{*} \langle |\text{comp}(c)|,\ \sigma'\rangle$, dann $\langle c,\ \sigma\rangle \Downarrow \sigma'$.

Für diesen Beweis benötigen wir noch den Begriff der Abgeschlossenheit. Das Verschiebungslemma erlaubte uns, beliebige Instruktionslisten zusammenzufügen. Für die Rückrichtung müssen wir Instruktionslisten an geeigneten Stellen aufteilen können. Dies ist aber nur möglich, wenn es keine Sprünge über die Aufteilungsgrenze gibt. Abgeschlossenheit ist ein ausreichendes Kriterium dafür:

Definition 33 (Abgeschlossenheit). Eine Instruktionsliste P heißt *abgeschlossen*, wenn alle Sprünge in P nur an die (absoluten) Positionen aus $\{0, \ldots, |P|\}$ gehen. Formal: Für alle $i < |P|$ mit $P_i = \texttt{JMP}\ n$ oder $P_i = \texttt{JMPF}\ n\ b$ gilt: $0 \le i + n \le |P|$.

Beispiel 34.
[JMPF 3 (x <= -1), ASSN x (-1 * x), JMP 1] ist abgeschlossen, nicht aber [JMPF 3 (x <= 5), ASSN x 17].

Lemma 35. $\text{comp}(c)$ ist abgeschlossen.

Beweis. Induktion über c. $\square$

Lemma 36 (Aufteilungslemma). Sei P abgeschlossen, $0 \le i \le |P|$ und $j \notin \{|P_0|, \ldots, |P_0| + |P| - 1\}$. Wenn $P_0 +\!\!+ P +\!\!+ P_1 \vdash \langle |P_0| + i,\ \sigma\rangle \xrightarrow{n} \langle j,\ \sigma'\rangle$, dann gibt es σ^*, n_1 und n_2 mit $n = n_1 + n_2$, $P \vdash \langle i,\ \sigma\rangle \xrightarrow{n_1} \langle |P|,\ \sigma^*\rangle$ und $P_0 +\!\!+ P +\!\!+ P_1 \vdash \langle |P_0| + |P|,\ \sigma^*\rangle \xrightarrow{n_2} \langle j,\ \sigma'\rangle$.

Beweis. Abkürzungen: $J = \{|P_0|, \ldots, |P_0| + |P| - 1\}$, $Q = P_0 +\!\!+ P +\!\!+ P_1$.
Induktion über n (σ und i beliebig):

- Basisfall $n = 0$: Zu zeigen: Wenn (i) $Q \vdash \langle |P_0| + i,\ \sigma\rangle \xrightarrow{0} \langle j,\ \sigma'\rangle$ und (ii) $0 \le i \le |P|$, dann gibt es σ^*, n_1 und n_2 mit $0 = n_1 + n_2$, $P \vdash \langle i,\ \sigma\rangle \xrightarrow{n_1} \langle |P|,\ \sigma^*\rangle$ und $Q \vdash \langle |P_0| + |P|,\ \sigma^*\rangle \xrightarrow{n_2} \langle j,\ \sigma'\rangle$.
  Aus (i) folgt $j = |P_0| + i$ und $\sigma' = \sigma$. Mit (ii) und $j \notin J$ folgt $i = |P|$. Damit folgt die Behauptung mit $n_1 = 0$, $n_2 = 0$ und $\sigma^* = \sigma$.
- Induktionsschritt $n + 1$:
  Induktionsannahme: Für alle $0 \le i \le |P|$ und σ gilt: Wenn $Q \vdash \langle |P_0| + i,\ \sigma\rangle \xrightarrow{n} \langle j,\ \sigma'\rangle$, dann gibt es σ^*, n_1 und n_2 mit $n = n_1 + n_2$, $P \vdash \langle i,\ \sigma\rangle \xrightarrow{n_1} \langle |P|,\ \sigma^*\rangle$ und $Q \vdash \langle |P_0| + |P|,\ \sigma^*\rangle \xrightarrow{n_2} \langle j,\ \sigma'\rangle$.
  Zu zeigen: Wenn (i) $Q \vdash \langle |P_0| + i,\ \sigma\rangle \xrightarrow{n+1} \langle j,\ \sigma'\rangle$ und $0 \le i \le |P|$, dann gibt es σ^*, n_1 und n_2 mit $n + 1 = n_1 + n_2$, $P \vdash \langle i,\ \sigma\rangle \xrightarrow{n_1} \langle |P|,\ \sigma^*\rangle$ und $Q \vdash \langle |P_0| + |P|,\ \sigma^*\rangle \xrightarrow{n_2} \langle j,\ \sigma'\rangle$.
  Wenn $i = |P|$, dann wähle $n_1 = 0$, $n_2 = n + 1$ und $\sigma^* = \sigma$.
  Sei also o.B.d.A (ii) $0 \le i < |P|$. Aus (i) gibt es i' und σ'' mit $Q \vdash \langle |P_0| + i,\ \sigma\rangle \to \langle i',\ \sigma''\rangle \xrightarrow{n} \langle j,\ \sigma\rangle$. Aus $Q \vdash \langle |P_0| + i,\ \sigma\rangle \to \langle i',\ \sigma''\rangle$ folgt (iii) $|P_0| \le i' \le |P_0| + |P|$ und (iv) $P \vdash \langle i,\ \sigma\rangle \to \langle i' - |P_0|,\ \sigma''\rangle$ durch Fallunterscheidung:
  - Fall ASSN: Damit $Q_{|P_0|+i} = \texttt{ASSN}\ x\ a$, $i' = |P_0| + i + 1$, $\sigma'' = \sigma[x \mapsto \mathcal{A}[\![a]\!]\,\sigma]$. Also $P_i = \texttt{ASSN}\ x\ a$ und $P \vdash \langle i,\ \sigma\rangle \to \langle i' - |P_0|,\ \sigma''\rangle$ nach Regel ASSN. Mit (ii) folgt $|P_0| \le i' \le |P_0| + |P|$.

– Fall Jmp: Damit $Q_{|P_0|+i} = \texttt{JMP}\ k = P_i$, $i' = |P_0| + i + k$, $\sigma'' = \sigma$. Somit $P \vdash \langle i,\ \sigma\rangle \to \langle i+k,\ \sigma''\rangle$ nach Regel Jmp. Da P abgeschlossen ist, gilt $0 \le i + k \le |P|$, somit $|P_0| \le i' \le |P_0| + |P|$.
– Fall JmpFT: Analog zu Fall Assn.
– Fall JmpFF: Analog zu Fall Jmp.

Aus (iii) folgt $0 \le i' - |P_0| \le |P|$. Damit ist die Induktionsannahme für $i = i' - |P_0|$ und $\sigma = \sigma''$ und $Q \vdash \langle i',\ \sigma''\rangle \overset{n}{\to} \langle j,\ \sigma'\rangle$ anwendbar und es gibt σ^*, n_1 und n_2 mit $n = n_1 + n_2$, (v) $P \vdash \langle i' - |P_0|,\ \sigma\rangle \overset{n_1}{\to} \langle |P|,\ \sigma^*\rangle$ und (vi) $Q \vdash \langle |P_0| + |P|,\ \sigma^*\rangle \overset{n_2}{\to} \langle j,\ \sigma'\rangle$. Aus (iv) und (v) folgt $P \vdash \langle i,\ \sigma\rangle \overset{n_1+1}{\to} \langle |P|,\ \sigma^*\rangle$. Mit (vi) und $n + 1 = (n_1 + 1) + n_2$ folgt die Behauptung. □

Theorem 37 (Big-Step simuliert ASM).
Wenn $\text{comp}(c) \vdash \langle 0,\ \sigma\rangle \overset{*}{\to} \langle |\text{comp}(c)|,\ \sigma'\rangle$, dann $\langle c,\ \sigma\rangle \Downarrow \sigma'$.

Beweis. Beweis durch Induktion über c (σ, σ' beliebig):

- Fall `skip`: Zu zeigen: Wenn (i) $[] \vdash \langle 0,\ \sigma\rangle \overset{*}{\to} \langle 0,\ \sigma'\rangle$, dann $\langle \texttt{skip},\ \sigma\rangle \Downarrow \sigma'$.
  Aus (i) folgt durch Regelinversion, dass $\sigma' = \sigma$, damit die Behauptung mit Regel Skip_{BS}.
- Fall x `:=` a: Zu zeigen: Wenn (i) $[\texttt{ASSN}\ x\ a] \vdash \langle 0,\ \sigma\rangle \overset{*}{\to} \langle 1,\ \sigma'\rangle$, dann $\langle x\ \texttt{:=}\ a,\ \sigma\rangle \Downarrow \sigma'$.
  Aus (i) folgt durch Regelinversion, dass $\sigma' = \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma]$. Damit gilt $\langle x\ \texttt{:=}\ a,\ \sigma\rangle \Downarrow \sigma'$ nach Regel Ass_{BS}.
- Fall c_1`;` c_2: Abkürzungen: $P = \text{comp}(c_1) +\!+ \text{comp}(c_2)$ und $l = |\text{comp}(c_1)| + |\text{comp}(c_2)|$
  Induktionsannahmen: Für alle σ und σ' gilt: Wenn $\text{comp}(c_1) \vdash \langle 0,\ \sigma\rangle \overset{*}{\to} \langle |\text{comp}(c_1)|,\ \sigma'\rangle$, dann $\langle c_1,\ \sigma\rangle \Downarrow \sigma'$; und wenn $\text{comp}(c_2) \vdash \langle 0,\ \sigma\rangle \overset{*}{\to} \langle |\text{comp}(c_2)|,\ \sigma'\rangle$, dann $\langle c_2,\ \sigma\rangle \Downarrow \sigma'$.
  Zu zeigen: Wenn (i) $P \vdash \langle 0,\ \sigma\rangle \overset{*}{\to} \langle l,\ \sigma'\rangle$, dann $\langle c_1\ \texttt{;}\ c_2,\ \sigma\rangle \Downarrow \sigma'$.

  Nach Lem. 35 sind $\text{comp}(c_1)$ und $\text{comp}(c_2)$ abgeschlossen. Nach dem Aufteilungslemma 36 und (i) gibt es ein σ^* mit (ii) $\text{comp}(c_1) \vdash \langle 0,\ \sigma\rangle \overset{*}{\to} \langle |\text{comp}(c_1)|,\ \sigma^*\rangle$ und (iii) $P \vdash \langle |\text{comp}(c_1)|,\ \sigma^*\rangle \overset{*}{\to} \langle l,\ \sigma'\rangle$.

  Aus (ii) folgt mit der Induktionsannahme, dass $\langle c_1,\ \sigma\rangle \Downarrow \sigma^*$.

  Wegen (iii) gibt es nach dem Aufteilungslemma ein σ^{**} mit (iv) $\text{comp}(c_2) \vdash \langle 0,\ \sigma^*\rangle \overset{*}{\to} \langle |\text{comp}(c_2)|,\ \sigma^{**}\rangle$ und (v) $P \vdash \langle l,\ \sigma^{**}\rangle \overset{*}{\to} \langle l,\ \sigma'\rangle$.

  Aus (iv) folgt mit der Induktionsannahme, dass $\langle c_2,\ \sigma^*\rangle \Downarrow \sigma^{**}$.

  Aus (v) folgt durch Regelinversion, dass $\sigma^{**} = \sigma'$ und damit die Behauptung mit Regel Seq_{BS}.
- Fall `if (`b`) then` c_1 `else` c_2: Abkürzungen: $l_1 = |\text{comp}(c_1)|$, $l_2 = |\text{comp}(c_2)|$,
  $P = [\texttt{JMPF}\ (l_1 + 2)\ b] +\!+ \text{comp}(c_1) +\!+ [\texttt{JMP}\ (l_2 + 1)] +\!+ \text{comp}(c_2)$.
  Induktionsannahmen: Für alle σ und σ' gilt: Wenn $\text{comp}(c_1) \vdash \langle 0,\ \sigma\rangle \overset{*}{\to} \langle |\text{comp}(c_1)|,\ \sigma'\rangle$, dann $\langle c_1,\ \sigma\rangle \Downarrow \sigma'$; und wenn $\text{comp}(c_2) \vdash \langle 0,\ \sigma\rangle \overset{*}{\to} \langle |\text{comp}(c_2)|,\ \sigma'\rangle$, dann $\langle c_2,\ \sigma\rangle \Downarrow \sigma'$.
  Zu zeigen: Wenn (i) $P \vdash \langle 0,\ \sigma\rangle \overset{*}{\to} \langle |l_1 + l_2 + 2|,\ \sigma'\rangle$, dann $\langle \texttt{if (}b\texttt{) then}\ c_1\ \texttt{else}\ c_2,\ \sigma\rangle \Downarrow \sigma'$.

  Beweis: Fallunterscheidung nach $\mathcal{B}[\![b]\!]\sigma$.
  – Fall $\mathcal{B}[\![b]\!]\sigma = \mathbf{tt}$: Aus (i) folgt $P \vdash \langle 0,\ \sigma\rangle \to \langle 1,\ \sigma\rangle \overset{*}{\to} \langle l_1 + l_2 + 2,\ \sigma'\rangle$ durch Regelinversion. Da $\text{comp}(c_1)$ abgeschlossen ist (Lem. 35), gibt es mit dem Aufteilungslemma 36 ein σ^* mit (i) $\text{comp}(c_1) \vdash \langle 0,\ \sigma\rangle \overset{*}{\to} \langle l_1,\ \sigma^*\rangle$ und (ii) $P \vdash \langle l_1 + 1,\ \sigma^*\rangle \overset{*}{\to} \langle l_1 + l_2 + 2,\ \sigma'\rangle$. Aus (i) folgt mit der Induktionsannahme, dass $\langle c_1,\ \sigma\rangle \Downarrow \sigma^*$.
    Aus (ii) und $P_{l_1+1} = \texttt{JMP}\ (l_2 + 1)$ folgt mit Regel-Inversion, dass $\sigma^* = \sigma'$. Zusammen mit $\mathcal{B}[\![b]\!]\sigma = \mathbf{tt}$ und $\langle c_1,\ \sigma\rangle \Downarrow \sigma^*$ folgt $\langle \texttt{if (}b\texttt{) then}\ c_1\ \texttt{else}\ c_2,\ \sigma\rangle \Downarrow \sigma'$ nach Regel IfTT_{BS}.
  – Fall $\mathcal{B}[\![b]\!]\sigma = \mathbf{ff}$: Analog für c_2 statt c_1.
- Fall `while (`b`) do` c: Abkürzungen: $P = \text{comp}(\texttt{while (}b\texttt{) do}\ c)$ und $l = |\text{comp}(c)|$.
  Induktionsannahme I: Wenn $\text{comp}(c) \vdash \langle 0,\ \sigma\rangle \overset{*}{\to} \langle l,\ \sigma'\rangle$, dann $\langle c,\ \sigma\rangle \Downarrow \sigma'$ für beliebige σ, σ'.
  Zu zeigen: Wenn $P \vdash \langle 0,\ \sigma\rangle \overset{*}{\to} \langle l + 2,\ \sigma'\rangle$, dann $\langle \texttt{while (}b\texttt{) do}\ c,\ \sigma\rangle \Downarrow \sigma'$.

  Beweis einer stärkeren Aussage durch vollständige Induktion über n (σ beliebig):
  Wenn $P \vdash \langle 0,\ \sigma\rangle \overset{n}{\to} \langle l + 2,\ \sigma'\rangle$, dann $\langle \texttt{while (}b\texttt{) do}\ c,\ \sigma\rangle \Downarrow \sigma'$.

Sei n beliebig. Induktionsannahme II:
Für alle $m < n$ und σ gilt: Wenn $P \vdash \langle 0,\ \sigma\rangle \overset{m}{\to} \langle l+2,\ \sigma'\rangle$, dann $\langle \texttt{while (}b\texttt{) do } c,\ \sigma\rangle \Downarrow \sigma'$.
Zu zeigen: Wenn (i) $P \vdash \langle 0,\ \sigma\rangle \overset{n}{\to} \langle l+2,\ \sigma'\rangle$, dann $\langle \texttt{while (}b\texttt{) do } c,\ \sigma\rangle \Downarrow \sigma'$.

Fallunterscheidung nach $\mathcal{B}[\![b]\!]\sigma$:

- Fall $\mathcal{B}[\![b]\!]\sigma = \mathbf{ff}$: Aus (i) folgt dann $n = 1$, $\sigma = \sigma'$ durch Regelinversion. Nach Regel $\textsc{WhileFF}_{\text{BS}}$ gilt $\langle \texttt{while (}b\texttt{) do } c,\ \sigma\rangle \Downarrow \sigma'$.
- Fall $\mathcal{B}[\![b]\!]\sigma = \mathbf{tt}$: Aus (i) folgt damit $n > 0$ und $P \vdash \langle 0,\ \sigma\rangle \to \langle 1,\ \sigma\rangle \overset{n-1}{\to} \langle l+2,\ \sigma'\rangle$.
  Da comp(c) abgeschlossen ist (Lem. 35), gibt es nach dem Aufteilungslemma 36 σ^*, n_1 und n_2 mit $n - 1 = n_1 + n_2$, (ii) comp(c) $\vdash \langle 0,\ \sigma\rangle \overset{n_1}{\to} \langle l,\ \sigma^*\rangle$ und (iii) $P \vdash \langle l+1,\ \sigma^*\rangle \overset{n_2}{\to} \langle l+2,\ \sigma'\rangle$.
  Aus (ii) folgt mit Induktionsannahme I, dass $\langle c,\ \sigma\rangle \Downarrow \sigma^*$.

  Aus (iii) und $P_{l+1} = \texttt{JMP } -(l+1)$ folgt durch Regelinversion: $P \vdash \langle l+1,\ \sigma^*\rangle \to \langle 0,\ \sigma^*\rangle \overset{n_2-1}{\to} \langle l+2,\ \sigma'\rangle$.

  Aus $P \vdash \langle 0,\ \sigma^*\rangle \overset{n_2-1}{\to} \langle l+2,\ \sigma'\rangle$ folgt mit der Induktionsannahme II für $m = n_2 - 1 < n$ und $\sigma = \sigma^*$, dass $\langle \texttt{while (}b\texttt{) do } c,\ \sigma^*\rangle \Downarrow \sigma'$.
  Zusammen mit $\langle c,\ \sigma\rangle \Downarrow \sigma^*$ und $\mathcal{B}[\![b]\!]\sigma = \mathbf{tt}$ folgt $\langle \texttt{while (}b\texttt{) do } c,\ \sigma\rangle \Downarrow \sigma'$ mit Regel $\textsc{WhileTT}_{\text{BS}}$. □

Aus diesem Theorem lässt sich nun „einfach“ die zweite Korrektheitsaussage des Compilers beweisen. Zuerst aber noch folgendes einfaches Hilfslemma:

Lemma 38. Sei P abgeschlossen.

(i) Wenn $P \vdash \langle i,\ \sigma\rangle \to \langle i',\ \sigma'\rangle$ und $0 \le i < |P|$, dann $0 \le i' \le |P|$.

(ii) Wenn $P \vdash \langle i,\ \sigma\rangle \overset{*}{\to} \langle i',\ \sigma'\rangle$ und $0 \le i \le |P|$, dann $0 \le i' \le |P|$.

Beweis. (i) durch Regelinversion, (ii) durch Induktion über $\overset{*}{\to}$, Induktionsschritt mit (i). □

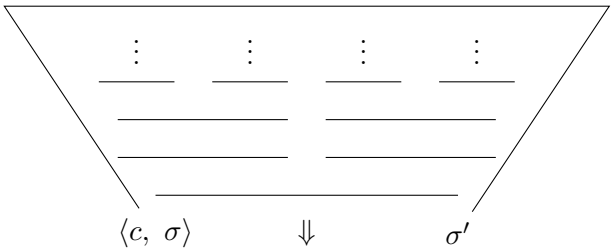
Korollar 39. Gibt es kein σ' mit $\langle c,\ \sigma\rangle \Downarrow \sigma'$, dann comp($c$) $\vdash \langle 0,\ \sigma\rangle \overset{\infty}{\to}$.

Beweis. Beweis durch Widerspruch. Angenommen, comp(c) $\vdash \langle 0,\ \sigma\rangle \overset{\infty}{\not\to}$. Dann gibt es i' und σ' mit comp(c) $\vdash \langle 0,\ \sigma\rangle \overset{*}{\to} \langle i',\ \sigma'\rangle \not\to$ Da comp(c) abgeschlossen ist (Lem. 35), folgt mit Lem. 38, dass $0 \le i' \le |\text{comp}(c)|$. Da $\langle i',\ \sigma'\rangle$ blockiert ist, gilt $i' = |\text{comp}(c)|$. Nach Thm. 37 gilt also $\langle c,\ \sigma\rangle \Downarrow \sigma'$. Widerspruch zur Voraussetzung. □

5.4 Vergleich der verschiedenen Semantiken

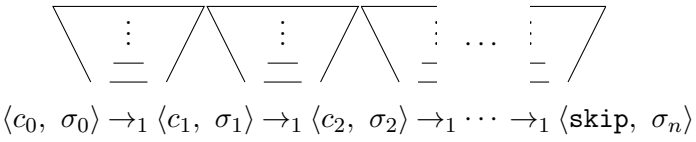
Big-Step-Semantik

- Das gesamte Programm wertet in einem Schritt zu einem Endzustand aus.
- Alle Berechnungsdetails sind in dem Ableitungsbaum versteckt.
- Beweisprinzip: Regelinduktion über die Auswertungsrelation.
- Nichttermination und Blockieren sind nicht unterscheidbar.



Small-Step-Semantik

- Ableitungsfolge enthält explizit alle Berechnungsschritte.
- Jeder Schritt wird durch einen einzelnen Ableitungsbaum gerechtfertigt.



- Beweisprinzip: Regelinduktion für einzelne Schritte und Induktion über die Länge der Ableitungsfolge
- Nichttermination und Blockieren ausdrückbar
- Die transitive, reflexive Hülle abstrahiert von den störenden, expliziten Zwischenschritten.

ASM

$$P \vdash \langle i_0,\ \sigma_0\rangle \to \langle i_1,\ \sigma_1\rangle \to \langle i_2,\ \sigma_2\rangle \to \ldots \to \langle |P|,\ \sigma_n\rangle$$

- Programm ist eine Liste, hat keine Baumstruktur.
- Ableitungsfolgen beschreiben wie bei einer Small-Step-Semantik die einzelnen Berechnungsschritte
- Jeder Schritt wird nur durch eine einzelne Regel gerechtfertigt.
- Beweisprinzip: Fallunterscheidung über die Instruktionen und Induktion über die Länge der Ableitungsfolge
- Nichttermination und Blockieren ausdrückbar.

6 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 40 (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.

6.1 Nichtdeterminismus While_{ND}

Sowohl Big-Step- als auch Small-Step-Semantik für While sind deterministisch (Thm. 10 und 17). 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 41. Das Programm `x := 5 or x := 7` kann der Variablen `x` entweder den Wert 5 oder den Wert 7 zuweisen.

6.1.1 Big-Step-Semantik

Die Ableitungsregeln für $\langle \_,\ \_\rangle \Downarrow \_$ werden für das neue Konstrukt c_1 `or` c_2 um die beiden folgenden erweitert:

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

Übung: Welche Ableitungsbäume hat das Programm $P \equiv$ `(x := 5) or (while (true) do skip)` in der Big-Step-Semantik?

6.1.2 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:

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

Beispiel 42. 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 nichtdeterministischen 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. 10 und 17)
- Fortschritt der Small-Step-Semantik (Lem. 16)
- Äquivalenz von Big-Step- und Small-Step-Semantik (Kor. 23)

6.2 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 43. 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.

6.3 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.

6.3.1 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 44. 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

6.3.2 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 45. Ableitungsfolge zu $P \equiv$ `{ var x = 0; { var y = 1; x := 5; y := x + y }; y := x }` im Startzustand $\sigma = [\mathtt{x} \mapsto 10, \mathtt{y} \mapsto 20]$:

$$\begin{aligned}
\langle P,\ \sigma\rangle &\to_1 \langle \texttt{\{ var x = 5; \{ var y = 1; y := x + y \}; y := x \}},\ \sigma\rangle \\
&\to_1 \langle \texttt{\{ var x = 5; \{ var y = 6; skip \}; y := x \}},\ \sigma\rangle \\
&\to_1 \langle \texttt{\{ var x = 5; skip; y := x \}},\ \sigma\rangle \to_1 \langle \texttt{\{ var x = 5; y := x \}},\ \sigma\rangle \\
&\to_1 \langle \texttt{\{ var x = 5; skip \}},\ \sigma[\mathtt{y} \mapsto 5]\rangle \to_1 \langle \texttt{skip},\ \sigma[\mathtt{y} \mapsto 5]\rangle
\end{aligned}$$

6.4 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]

6.4.1 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 und die `try`-Anweisung.

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

Lemma 46 (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 16 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. 16)
    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 $\langle \texttt{try } c_1 \texttt{ catch } Y\ c_2,\ \sigma\rangle \to_1 \langle \texttt{try } c_1' \texttt{ catch } Y\ c_2,\ \sigma'\rangle$. □

6.4.2 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

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

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:

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

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

$$\textsc{IfTT}_{BS}\colon \frac{\mathcal{B}[\![b]\!]\,\sigma = \mathbf{tt} \qquad \langle c_0,\ \sigma\rangle \Downarrow (\xi, \sigma')}{\langle \texttt{if (}b\texttt{) then}\ c_0\ \texttt{else}\ c_1,\ \sigma\rangle \Downarrow (\xi, \sigma')}$$

$$\textsc{IfFF}_{BS}\colon \frac{\mathcal{B}[\![b]\!]\,\sigma = \mathbf{ff} \qquad \langle c_1,\ \sigma\rangle \Downarrow (\xi, \sigma')}{\langle \texttt{if (}b\texttt{) then}\ c_0\ \texttt{else}\ c_1,\ \sigma\rangle \Downarrow (\xi, \sigma')}$$

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

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

Regeln für die neuen Konstrukte:

$$\textsc{Raise}\colon \langle \texttt{raise}\ X,\ \sigma\rangle \Downarrow (X, \sigma)$$

$$\textsc{Try}_{BS}\colon \frac{\langle c_1,\ \sigma\rangle \Downarrow (\xi, \sigma') \qquad \xi \neq X}{\langle \texttt{try}\ c_1\ \texttt{catch}\ X\ c_2,\ \sigma\rangle \Downarrow (\xi, \sigma')}$$

$$\textsc{Catch}\colon \frac{\langle c_1,\ \sigma\rangle \Downarrow (X, \sigma') \qquad \langle c_2,\ \sigma'\rangle \Downarrow (\xi, \sigma'')}{\langle \texttt{try}\ c_1\ \texttt{catch}\ X\ c_2,\ \sigma\rangle \Downarrow (\xi, \sigma'')}$$

Zusätzliche Regeln mit Abbruch für alle sequenz-ähnlichen Konstrukte:

$$\textsc{SeqX}_{BS}\colon \frac{\langle c_0,\ \sigma\rangle \Downarrow (X, \sigma')}{\langle c_0\texttt{;}\ c_1,\ \sigma\rangle \Downarrow (X, \sigma')} \qquad \textsc{WhileTTX}_{BS}\colon \frac{\mathcal{B}[\![b]\!]\,\sigma = \mathbf{tt} \qquad \langle c,\ \sigma\rangle \Downarrow (X, \sigma')}{\langle \texttt{while (}b\texttt{) do}\ c,\ \sigma\rangle \Downarrow (X, \sigma')}$$

Diese Modellierung für Exceptions führt dazu, dass man für jede Regel, die in ihren Annahmen mehrere Programme nacheinander auswertet, entsprechend oft dupliziert werden muss, das kann schnell unhandlich werden.

Variation Diese Problem kann umgangen werden, wenn das Exception-Flag auch auf der linken Seite von $\Downarrow$ vorkommen kann:

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

Dabei ist darauf zu achten dass alle Regeln nur dann anwendbar sind, wenn keine Exception vorliegt:

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

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

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

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

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

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

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

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

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

Zusätzlich braucht es nun eine Regel, die bei gesetztem Exception-Flag angewandt werden kann, und in dem Fall das Programm ignoriert und die Exception weitergibt:

$$\textsc{Propagate}\colon \langle c,\ (X, \sigma)\rangle \Downarrow (X, \sigma)$$

Die beiden Varianten der Definition sind äquivalent:

$$\langle c,\ \sigma\rangle \Downarrow (\xi, \sigma') \iff \langle c,\ (\texttt{None}, \sigma)\rangle \Downarrow (\xi, \sigma')$$

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

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

Die Beweise verlaufen ähnlich wie für Thm. 21 und 22.

6.5 Prozeduren

In diesem Abschnitt erweitern wir die While_B-Sprache um Prozeduren bzw. Funktionen. Waren die bisherigen Semantiken für While mit Erweiterungen meist ohne große Designentscheidungen, so gibt es bei Prozeduren mehrere Modellierungsmöglichkeiten mit unterschiedlicher Semantik. Wir beginnen mit der einfachsten Form, bei der Prozeduren quasi textuell an die Aufrufstelle kopiert werden und ohne Parameter auskommen, ähnlich zu Makros.[3]

6.5.1 Prozeduren ohne Parameter While_{PROC}

Die Syntax von While_{PROC} muss dazu Deklarationsmöglichkeiten für und Aufrufe von Prozeduren bereitstellen. Ein Programm P besteht ab sofort aus einer Anweisung c und einer Liste von Prozedurdeklarationen der Form (p, c), wobei p den Namen der Prozedur und c den Rumpf der Prozedur beschreibt. Wir nehmen im Folgenden immer an, dass die Prozedurnamen in der Deklarationsliste eindeutig sind. Die neue Anweisung `call` p ruft die Prozedur p auf.

Beispiel 47. ($\texttt{sum}$, `if (i == 0) then skip else (x := x + i; i := i - 1; call sum)`) deklariert eine Prozedur `sum`. Damit berechnet `x := 0; call sum` die Summe der ersten $\sigma(\texttt{i})$ Zahlen, falls $\sigma(\texttt{i}) \geq 0$ ist.

Wenden wir uns nun als erstes der Big-Step-Semantik zu. Diese braucht für die Aufrufregel die Prozedurdeklarationen. Deswegen ändern wir den Typ der Big-Step-Auswertungsrelation so, dass die Deklarationen als eine Umgebung P durch alle Regeln durchgeschleift werden:

$$\_ \vdash \langle \_,\ \_ \rangle \Downarrow \_ \ \subseteq\ \mathsf{PDecl}^* \times (\mathsf{Com} \times \Sigma) \times \Sigma$$

Entsprechend müssen alle bisherigen Regeln angepasst werden:

$$\textsc{Skip}^{\mathrm{P}}_{\mathrm{BS}}\colon\ P \vdash \langle \texttt{skip},\ \sigma \rangle \Downarrow \sigma \qquad \textsc{Ass}^{\mathrm{P}}_{\mathrm{BS}}\colon\ P \vdash \langle x\ \texttt{:=}\ a,\ \sigma \rangle \Downarrow \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma]$$

$$\textsc{Seq}^{\mathrm{P}}_{\mathrm{BS}}\colon\ \frac{P \vdash \langle c_0,\ \sigma \rangle \Downarrow \sigma' \qquad P \vdash \langle c_1,\ \sigma' \rangle \Downarrow \sigma''}{P \vdash \langle c_0\texttt{;}\ c_1,\ \sigma \rangle \Downarrow \sigma''}$$

$$\textsc{IfTT}^{\mathrm{P}}_{\mathrm{BS}}\colon\ \frac{\mathcal{B}[\![b]\!]\sigma = \mathbf{tt} \qquad P \vdash \langle c_0,\ \sigma \rangle \Downarrow \sigma'}{P \vdash \langle \texttt{if (}b\texttt{) then}\ c_0\ \texttt{else}\ c_1,\ \sigma \rangle \Downarrow \sigma'}$$

$$\textsc{IfFF}^{\mathrm{P}}_{\mathrm{BS}}\colon\ \frac{\mathcal{B}[\![b]\!]\sigma = \mathbf{ff} \qquad P \vdash \langle c_1,\ \sigma \rangle \Downarrow \sigma'}{P \vdash \langle \texttt{if (}b\texttt{) then}\ c_0\ \texttt{else}\ c_1,\ \sigma \rangle \Downarrow \sigma'}$$

$$\textsc{WhileFF}^{\mathrm{P}}_{\mathrm{BS}}\colon\ \frac{\mathcal{B}[\![b]\!]\sigma = \mathbf{ff}}{P \vdash \langle \texttt{while (}b\texttt{) do}\ c,\ \sigma \rangle \Downarrow \sigma}$$

$$\textsc{WhileTT}^{\mathrm{P}}_{\mathrm{BS}}\colon\ \frac{\mathcal{B}[\![b]\!]\sigma = \mathbf{tt} \qquad P \vdash \langle c,\ \sigma \rangle \Downarrow \sigma' \qquad P \vdash \langle \texttt{while (}b\texttt{) do}\ c,\ \sigma' \rangle \Downarrow \sigma''}{P \vdash \langle \texttt{while (}b\texttt{) do}\ c,\ \sigma \rangle \Downarrow \sigma''}$$

$$\textsc{Block}^{\mathrm{P}}_{\mathrm{BS}}\colon\ \frac{P \vdash \langle c,\ \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma] \rangle \Downarrow \sigma'}{P \vdash \langle \texttt{\{ var}\ x\ \texttt{=}\ a\texttt{;}\ c\ \texttt{\}},\ \sigma \rangle \Downarrow \sigma'[x \mapsto \sigma(x)]}$$

[3] Makros werden üblicherweise durch einen Präprozessor *statisch* im Text ersetzt, der Compiler oder die Semantik sehen von den Makros selbst nichts. Unsere Prozeduren dagegen werden erst *zur Laufzeit* eingesetzt und können dadurch beliebig rekursiv sein.

Außerdem brauchen wir noch eine neue Regel für den Prozeduraufruf:

$$\textsc{Call}^{\text{P}}_{\text{BS}}\colon \frac{(p, c) \in P \qquad P \vdash \langle c,\ \sigma\rangle \Downarrow \sigma'}{P \vdash \langle \texttt{call}\ p,\ \sigma\rangle \Downarrow \sigma'}$$

Die Small-Step-Semantik ist für diese Variante der Prozeduren genauso einfach. Wie bei der Big-Step-Semantik erweitern wir $\langle \_,\ \_\rangle \to_1 \langle \_,\ \_\rangle$ um die Prozedurdeklarationsumgebung P, die Regeln wiederholen wir hier nicht nochmals. Für die Aufrufanweisung `call` p ergibt sich folgende Regel:

$$\textsc{Call}^{\text{P}}_{\text{SS}}\colon \frac{(p, c) \in P}{P \vdash \langle \texttt{call}\ p,\ \sigma\rangle \to_1 \langle c,\ \sigma\rangle}$$

Beispiel 48. Sei P die Liste, die nur die Prozedur `sum` wie im letzten Beispiel deklariert. Der Ableitungsbaum für `call sum` im Anfangszustand $\sigma_1 = [\mathtt{i} \mapsto 2, \mathtt{x} \mapsto 0]$ ist:

$\sigma_1 = [\mathtt{i} \mapsto 2, \mathtt{x} \mapsto 0]$
$\sigma_2 = [\mathtt{i} \mapsto 1, \mathtt{x} \mapsto 2]$
$\sigma_3 = [\mathtt{i} \mapsto 0, \mathtt{x} \mapsto 3]$

$$\dfrac{(\mathtt{sum}, c_{\mathtt{sum}}) \in P \qquad \dfrac{\dots \qquad \dfrac{\dots \qquad \dfrac{\dots \qquad \dfrac{\dots \qquad \dfrac{}{P \vdash \langle \mathtt{skip},\ \sigma_3\rangle \Downarrow \sigma_3}}{P \vdash \langle c_{\mathtt{sum}},\ \sigma_3\rangle \Downarrow \sigma_3}}{P \vdash \langle \mathtt{call\ sum},\ \sigma_3\rangle \Downarrow \sigma_3}}{P \vdash \langle c_{\mathtt{else}},\ \sigma_2\rangle \Downarrow \sigma_3}}{P \vdash \langle c_{\mathtt{sum}},\ \sigma_2\rangle \Downarrow \sigma_3} \qquad \dots}{P \vdash \langle \mathtt{call\ sum},\ \sigma_2\rangle \Downarrow \sigma_3} \qquad \dots}{P \vdash \langle c_{\mathtt{else}},\ \sigma_1\rangle \Downarrow \sigma_3} \qquad (\mathtt{sum}, c_{\mathtt{sum}}) \in P}{P \vdash \langle c_{\mathtt{sum}},\ \sigma_1\rangle \Downarrow \sigma_3}}{P \vdash \langle \mathtt{call\ sum},\ \sigma_1\rangle \Downarrow \sigma_3}$$

Übung: Welche Ableitungsfolge ergibt sich in der Small-Step-Semantik für dieses Beispiel?

Obwohl die Semantik-Regeln einfach sind, ist diese Modellierung von Prozeduren aus folgenden Gründen nicht zufriedenstellend:

1. Es gibt keine Parameter, Wertüber- und -rückgabe ist nur über globale Variablen möglich.
2. Die Variablen in der Prozedur sind nicht statisch an globale Variablen gebunden, sondern werden *dynamisch* gebunden. Wenn ein umgebender Block eine lokale Variable deklariert, deren Namen in der Prozedur verwendet wird, so arbeitet die Prozedur beim Aufruf innerhalb des Blocks mit der lokalen Variable, außerhalb aber mit der globalen. Damit werden Abstraktionen unmöglich.

6.5.2 Prozeduren mit einem Parameter While$_{PROCP}$

Wegen der obigen Nachteile wollen wir nun noch eine Modellierung mit expliziten Parametern und statisch gebundenen Variablen ausarbeiten. Diese Änderung ist nicht modular, weil wir dafür die Zustandsmodellierung ändern müssen, so dass Variablen je nach Bedarf an andere Speicherstellen gebunden werden können.

Definition 49 (Speicher, Variablenumgebung). Der *Speicher* (*store*) s ist eine Funktion von *Speicherstellen* (*locations*) auf Werte. Eine *Variablenumgebung* E ordnet jeder Variablen eine Speicherstelle (location) zu, hat also den Typ $\mathsf{Var} \Rightarrow \mathsf{Loc}$.

Ein Zugriff auf eine Variable x erfolgt nun dadurch, dass

1. Die der Variablen x zugeordnete Speicherstelle $E(x)$ ermittelt wird, und
2. Im Speicher s auf die Stelle $E(x)$ – mit dem gespeicherten Wert $s(E(x))$ – zugegriffen wird.

Der Einfachheit halber sei $\mathsf{Loc} = \mathbb{Z}$. Neben der Speicherung der Werte der Speicherstellen muss ein Speicher auch noch vermerken, welche Speicherstelle die nächste unbenutzte ist. Demnach ist ein Speicher vom Typ

$$\mathsf{Store} \equiv \mathsf{Loc} + \{\mathsf{next}\} \Rightarrow \mathbb{Z},$$

wobei unter next die nächste freie Speicherzelle vermerkt ist.[4]

Definition 50 (Programm). Ein Programm der neuen Sprache While_{PROCP} besteht aus

1. einer Liste P von Prozedurdeklarationen,
2. der auszuführenden Anweisung und
3. einer Liste V der globalen Variablen, die von den Prozeduren und der Anweisung verwendet werden.

Definition 51 (Initiale Variablenumgebung, initialer Zustand). Die initiale Variablenumgebung E_0 ordnet den globalen Variablen die ersten $|V|$ Speicherstellen, d.h. von 0 bis $|V| - 1$, zu. Der initiale Zustand muss unter next die nächste freie Speicherstelle $|V|$ speichern.

Da Prozeduren nun auch einen Parameter bekommen und einen Rückgabewert berechnen sollen, müssen auch Prozedurdeklarationen und Aufrufe angepasst werden. Konzeptuell kann unser Ansatz auch auf mehrere Parameter- oder Rückgabewerte erweitert werden. Wegen der zusätzlichen formalen Komplexität betrachten wir hier aber nur Prozeduren mit einem Parameter.

Definition 52 (Prozedurdeklaration). Eine Prozedurdeklaration besteht nun aus

1. dem Prozedurnamen p,
2. dem Parameternamen x und
3. dem Rumpf der Prozedur als Anweisung.

Den Rückgabewert muss jede Prozedur in die spezielle (prozedurlokale) Variable `result` schreiben. Damit hat jede Prozedur automatisch zwei lokale Variablen: Den Parameter und die Rückgabevariable `result`.

Ein Aufruf hat nun die Form y `<- call` $p(a)$, wobei p der Prozedurname, a der arithmetische Ausdruck, dessen Wert an den Parameter übergeben wird und y die Variable ist, die den Rückgabewert aufnimmt.

Beispiel 53. Gegeben sei die Deklaration der Prozedur `sum2` mit Parameter `i` und Rumpf
`if (i == 0) then result := 0 else (result <- call sum2(i - 1); result := result + i)`
Der Aufruf `x <- call sum2(10)` speichert in der Variablen x die Summe der ersten 10 Zahlen.

Für die Big-Step-Semantik muss die Auswertungsrelation wieder erweitert werden: Wir brauchen zusätzlich die globale Umgebung E_0 und die aktuelle Umgebung E, die den Variablen Speicherstellen

[4] Da wir $\mathsf{Loc} = \mathbb{Z}$ gewählt haben, genügt uns dieser einfache Typ, da $s(\mathsf{next}) \in \mathsf{Loc}$ gelten muss. Im allgemeinen Fall wäre $\mathsf{Store} \equiv (\mathsf{Loc} \Rightarrow \mathbb{Z}) \times \mathsf{Loc}$, was die Syntax aufwändiger machte.

zuordnen. Zum Programmstart sind diese beiden gleich, im Laufe der Ausführung kann sich E aber ändern. Außerdem gibt es keinen Zustand σ mehr, sondern nur noch einen globalen Speicher s. Damit hat die Auswertungsrelation folgende Form:

$$P, E_0, E \vdash \langle c,\ s\rangle \Downarrow s'$$

Wie schon mit P geschehen, müssen die Umgebungen E_0 und E durch alle Regeln durchgeschleift werden. Variablenzugriffe müssen jetzt über E und s erfolgen.

Definition 54 (Big-Step-Semantik für Prozeduren mit einem Parameter).
Die geänderten Regeln für die Auswertungsrelation sehen wie folgt aus:

$$\textsc{Skip}^{\text{P1}}_{\text{BS}}\colon P, E_0, E \vdash \langle \texttt{skip},\ s\rangle \Downarrow s \qquad \textsc{Ass}^{\text{P1}}_{\text{BS}}\colon P, E_0, E \vdash \langle x\ \texttt{:=}\ a,\ s\rangle \Downarrow s[E(x) \mapsto \mathcal{A}[\![a]\!](s \circ E)]$$

$$\textsc{Seq}^{\text{P1}}_{\text{BS}}\colon \frac{P, E_0, E \vdash \langle c_0,\ s\rangle \Downarrow s' \qquad P, E_0, E \vdash \langle c_1,\ s'\rangle \Downarrow s''}{P, E_0, E \vdash \langle c_0\,\texttt{;}\ c_1,\ s\rangle \Downarrow s''}$$

$$\textsc{IfTT}^{\text{P1}}_{\text{BS}}\colon \frac{\mathcal{B}[\![b]\!](s \circ E) = \mathbf{tt} \qquad P, E_0, E \vdash \langle c_0,\ s\rangle \Downarrow s'}{P, E_0, E \vdash \langle \texttt{if (}b\texttt{) then } c_0 \texttt{ else } c_1,\ s\rangle \Downarrow s'}$$

$$\textsc{IfFF}^{\text{P1}}_{\text{BS}}\colon \frac{\mathcal{B}[\![b]\!](s \circ E) = \mathbf{ff} \qquad P, E_0, E \vdash \langle c_1,\ s\rangle \Downarrow s'}{P, E_0, E \vdash \langle \texttt{if (}b\texttt{) then } c_0 \texttt{ else } c_1,\ s\rangle \Downarrow s'}$$

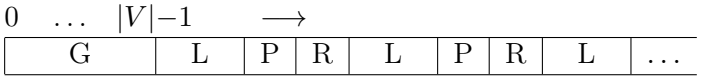
$$\textsc{WhileFF}^{\text{P1}}_{\text{BS}}\colon \frac{\mathcal{B}[\![b]\!](s \circ E) = \mathbf{ff}}{P, E_0, E \vdash \langle \texttt{while (}b\texttt{) do } c,\ s\rangle \Downarrow s}$$

$$\textsc{WhileTT}^{\text{P1}}_{\text{BS}}\colon \frac{\mathcal{B}[\![b]\!](s \circ E) = \mathbf{tt} \qquad P, E_0, E \vdash \langle c,\ s\rangle \Downarrow s' \qquad P, E_0, E \vdash \langle \texttt{while (}b\texttt{) do } c,\ s'\rangle \Downarrow s''}{P, E_0, E \vdash \langle \texttt{while (}b\texttt{) do } c,\ s\rangle \Downarrow s''}$$

$$\textsc{Block}^{\text{P1}}_{\text{BS}}\colon \frac{P, E_0, E[x \mapsto s(\mathsf{next})] \vdash \langle c,\ s[s(\mathsf{next}) \mapsto \mathcal{A}[\![a]\!](s \circ E), \mathsf{next} \mapsto s(\mathsf{next}) + 1]\rangle \Downarrow s'}{P, E_0, E \vdash \langle \texttt{\{ var } x \texttt{ = } a\texttt{; } c \texttt{ \}},\ s\rangle \Downarrow s'[\mathsf{next} \mapsto s(\mathsf{next})]}$$

$$\textsc{Call}^{\text{P1}}_{\text{BS}}\colon \frac{(p, x, c) \in P \qquad \begin{array}{c} P, E_0, E_0[x \mapsto s(\mathsf{next}), \texttt{result} \mapsto s(\mathsf{next}) + 1] \vdash \\ \langle c,\ s[s(\mathsf{next}) \mapsto \mathcal{A}[\![a]\!](s \circ E), \mathsf{next} \mapsto s(\mathsf{next}) + 2]\rangle \Downarrow s' \end{array}}{P, E_0, E \vdash \langle y\ \texttt{<- call } p\texttt{(}a\texttt{)},\ s\rangle \Downarrow s'[E(y) \mapsto s'(s(\mathsf{next}) + 1), \mathsf{next} \mapsto s(\mathsf{next})]}$$

Die Regeln $\textsc{Block}^{\text{P1}}_{\text{BS}}$ und $\textsc{Call}^{\text{P1}}_{\text{BS}}$ allozieren nun explizit neuen Speicher für die lokale Variable bzw. den Parameter und `result`. Nach der Ausführung setzen sie den next-Zeiger auf den Wert vor Beginn der Ausführung zurück. Dies ist möglich, weil neue Variablen (und damit neuer Speicher) nur strukturiert durch Blöcke bzw. Prozeduraufrufe alloziert werden, d.h., der Speicher wird stack-artig verwendet. Anschaulich ergibt sich folgende Speicheraufteilung:



wobei

G	globale Variablen
L	lokale Variablen
P	Parameter
R	Rückgabewert `result`

Beispiel 55. Sei P die Prozedurliste, die nur die Prozedur `sum2` vom letzten Beispiel deklariert, und das Hauptprogramm $c \equiv$ `x <- call sum2(2)`. Die Liste V der globalen Variablen ist dann [x]. Damit ergibt sich die initiale Variablenumgebung E_0 zu $[\mathtt{x} \mapsto 0]$ und der Anfangsspeicher $s_0 \equiv [0 \mapsto ?, \mathsf{next} \mapsto 1]$, wobei der Anfangswert von `x` einen beliebigen Wert ? hat. Der Ableitungsbaum für c ist:

$$\dfrac{(\texttt{sum2}, \texttt{i}, c_{\texttt{sum2}}) \in P \qquad \dfrac{\mathcal{B}[\![\texttt{i == 0}]\!](s_1 \circ E_1) = \mathbf{ff} \qquad \dfrac{A \qquad \dfrac{}{P, E_0, E_1 \vdash \langle \texttt{result := result + i},\ s_7\rangle \Downarrow s_8}}{P, E_0, E_1 \vdash \langle c_{\texttt{else}},\ s_1\rangle \Downarrow s_8}}{P, E_0, E_1 \vdash \langle c_{\texttt{sum2}},\ s_1\rangle \Downarrow s_8}}{P, E_0, E_0 \vdash \langle c,\ s_0\rangle \Downarrow s_9}$$

$$\text{A:}\ \dfrac{(\texttt{sum2}, \texttt{i}, c_{\texttt{sum2}}) \in P \qquad \dfrac{\mathcal{B}[\![\texttt{i == 0}]\!](s_2 \circ E_2) = \mathbf{ff} \qquad \dfrac{B \qquad \dfrac{}{P, E_0, E_2 \vdash \langle \texttt{result := result + i},\ s_5\rangle \Downarrow s_6}}{P, E_0, E_2 \vdash \langle c_{\texttt{else}},\ s_2\rangle \Downarrow s_6}}{P, E_0, E_2 \vdash \langle c_{\texttt{sum2}},\ s_2\rangle \Downarrow s_6}}{P, E_0, E_1 \vdash \langle \texttt{result <- call sum2(i - 1)},\ s_1\rangle \Downarrow s_7}$$

$$\text{B:}\ \dfrac{(\texttt{sum2}, \texttt{i}, c_{\texttt{sum2}}) \in P \qquad \dfrac{\mathcal{B}[\![\texttt{i == 0}]\!](s_3 \circ E_3) = \mathbf{tt} \qquad \dfrac{}{P, E_0, E_3 \vdash \langle \texttt{result := 0},\ s_3\rangle \Downarrow s_4}}{P, E_0, E_3 \vdash \langle c_{\texttt{sum2}},\ s_3\rangle \Downarrow s_4}}{P, E_0, E_2 \vdash \langle \texttt{result <- call sum2(i - 1)},\ s_2\rangle \Downarrow s_5}$$

wobei

Variablenumgebung:	**Belegung:**	x	i	result
$E_0 = [\texttt{x} \mapsto 0]$		0		
$E_1 = E_0[\texttt{i} \mapsto s_0(\mathsf{next}), \texttt{result} \mapsto s_0(\mathsf{next}) + 1]$		0	1	2
$E_2 = E_0[\texttt{i} \mapsto s_1(\mathsf{next}), \texttt{result} \mapsto s_1(\mathsf{next}) + 1]$		0	3	4
$E_3 = E_0[\texttt{i} \mapsto s_2(\mathsf{next}), \texttt{result} \mapsto s_2(\mathsf{next}) + 1]$		0	5	6

Speicher:	**Werte:**	next	0	1	2	3	4	5	6
$s_0 = [0 \mapsto ?, \mathsf{next} \mapsto 1]$		1	?						
$s_1 = s_0[1 \mapsto \mathcal{A}[\![2]\!](s_0 \circ E_0), \mathsf{next} \mapsto 3]$		3	?	2	?				
$s_2 = s_1[3 \mapsto \mathcal{A}[\![\texttt{i - 1}]\!](s_1 \circ E_1), \mathsf{next} \mapsto 5]$		5	?	2	?	1	?		
$s_3 = s_2[5 \mapsto \mathcal{A}[\![\texttt{i - 1}]\!](s_2 \circ E_2), \mathsf{next} \mapsto 7]$		7	?	2	?	1	?	0	?
$s_4 = s_3[E_3(\texttt{result}) \mapsto \mathcal{A}[\![0]\!](s_3 \circ E_3)]$		7	?	2	?	1	?	0	0
$s_5 = s_4[E_2(\texttt{result}) \mapsto s_5(s_2(\mathsf{next}) + 1), \mathsf{next} \mapsto 5]$		5	?	2	?	1	0	*0*	*0*
$s_6 = s_5[E_2(\texttt{result}) \mapsto \mathcal{A}[\![\texttt{result + i}]\!](s_5 \circ E_2)]$		5	?	2	?	1	1	*0*	*0*
$s_7 = s_6[E_1(\texttt{result}) \mapsto s_6(s_1(\mathsf{next}) + 1), \mathsf{next} \mapsto 3]$		3	?	2	1	*1*	*1*	*0*	*0*
$s_8 = s_7[E_1(\texttt{result}) \mapsto \mathcal{A}[\![\texttt{result + i}]\!](s_7 \circ E_1)]$		3	?	2	3	*1*	*1*	*0*	*0*
$s_9 = s_8[E_0(\texttt{x}) \mapsto s_8(s_0(\mathsf{next}) + 1), \mathsf{next} \mapsto 1]$		1	3	*2*	*3*	*1*	*1*	*0*	*0*

6.6 Getypte Variablen While$_T$

Bisher speicherten alle Variablen in unseren Programmen ausschließlich ganze Zahlen, aber keine booleschen Werte. Wir ändern While$_B$ jetzt, sodass Zuweisungen auch für boolesche Werte erlaubt sind, und nennen die neue Sprache While$_T$. Dabei stößt man aber schnell auf ein Problem: Welchen Wert hat `y` am Ende des Programms

```
x := true; y := x + 5?
```

Genau genommen möchten wir dieses Programm als ungültig zurückweisen. Im Allgemeinen ist es aber sehr schwierig (bzw. unentscheidbar), ob ein Programm in diesem Sinn als gültig anzusehen ist.

Übung: Welche der folgenden Programme sollten Ihrer Meinung nach als ungültig zurückgewiesen werden?

- `x := true; x := 0`
- `y := true; (if (y) then x := true else x := 0); z := x`
- `x := true; { var x = 0; y := x }; y := y + 2`

Typen legen die möglichen Wertebereiche für Variablen (und damit auch Ausdrücke) fest, in unserem Fall ganze Zahlen bzw. Wahrheitswerte. Ein (statisches) *Typsystem* ist eine spezielle Form der Programmanalyse bzw. -verifikation, die die Typen eines Programms „zur Übersetzungszeit“ automatisch (und effizient) analysiert. Dabei wird das Programm insbesondere *nicht* ausgeführt. Das Typsystem garantiert, dass bestimmte Laufzeitfehler in typkorrekten Programmen nicht auftreten können, beispielsweise, dass die Operanden einer arithmetischen Operation wirklich Zahlen (und keine Wahrheitswerte) sind. Viele Programmier- und Tippfehler zeigen sich bereits durch Typfehler: „Well-typed programs cannot go wrong“ [R. Milner]. Aber nicht alle Laufzeitfehler können mittels (einfachen) Typsystemen ausgeschlossen werden, z.B. Division durch 0.

6.6.1 Typen für While$_T$

Wir unterscheiden ab jetzt arithmetische und boolesche Ausdrücke nicht mehr syntaktisch, das Typsystem wird sich später um die Trennung kümmern.

Definition 56 (Ungetypte Ausdrücke). Die Syntax für (ungetypte) Ausdrücke Exp lautet:

$$\mathsf{Exp}\quad e \quad ::= \quad n \mid x \mid e_1 \texttt{ - } e_2 \mid e_1 \texttt{ * } e_2 \mid \texttt{true} \mid e_1 \texttt{ <= } e_2 \mid \texttt{not } b \mid b_1 \texttt{ \&\& } b_2$$

Entsprechend passt sich auch die Syntax für Anweisungen an: Wo bisher arithmetische oder boolesche Ausdrücke gefordert waren, verlangen wir nur noch (ungetypte) Ausdrücke:

Definition 57 (Syntax für Anweisungen in While$_T$).

$$\begin{aligned}\mathsf{Com}\quad c \quad ::= \quad & \texttt{skip} \mid x \texttt{ := } e \mid c_1\texttt{; } c_2 \mid \texttt{if (}e\texttt{) then } c_1 \texttt{ else } c_2 \mid \texttt{while (}e\texttt{) do } c \mid \\ & \texttt{\{ var } x \texttt{ = } e\texttt{; } c \texttt{ \}}\end{aligned}$$

Definition 58 (Typen). int und bool sind die Typen $\mathbb{T}$ für While$_T$:

$$\mathbb{T} = \{\mathsf{int},\ \mathsf{bool}\}$$

Nach Variablenkonvention steht die Metavariable τ immer für einen Typen aus $\mathbb{T}$.

Die Annahme für unser statisches Typsystem lautet: Jede Variable hat (innerhalb ihres Gültigkeitsbereichs) einen festen Typ. Damit brauchen wir uns keine Gedanken darüber zu machen, ob eine Variable an Kontrollzusammenflussstellen (z.B. nach einer Fallunterscheidung) immer den gleichen Typ hat, egal wie man an diese Stelle gelangt ist.

Für die Typkorrektheitsüberprüfung gibt es drei Ansätze:

1. Die Sprachdefinition legt fest, welche Variablen für Zahlen und welche für Wahrheitswerte verwendet werden dürfen. Dies kommt jedoch wieder einer syntaktischen Trennung arithmetischer und boolescher Ausdrücke gleich und in keiner modernen Programmiersprache mehr vor. In Fortran haben standardmäßig alle Variablen mit den Anfangsbuchstaben i, j, k, l, m oder n den Typ Integer und alle anderen den Typ Real.
2. Jedes Programm deklariert die Typen der Variablen, die es verwendet. Das Typsystem führt also lediglich eine *Typüberprüfung* durch.
3. Der Typ wird aus den Verwendungen der Variablen erschlossen. Das Typsystem führt eine *Typinferenz* durch. Bei Typinferenz tritt in der Praxis, insbesondere wenn Prozeduren und Funktionen involviert sind, oft das Problem auf, dass Fehlermeldungen des Typcheckers nur schwer zu verstehen sind, weil der Typfehler erst an einer ganz anderen Stelle auftritt, als der eigentliche Tippfehler des Programmierers ist.

Beispiel 59. Für die Anweisung `b := c; i := 42 + j` könnten diese drei Ansätze wie folgt aussehen:

1. Die Sprachdefinition legt fest, dass Variablen, deren Namen mit `i` und `j` beginnen, in *allen* Programmen immer den Typ int haben, und dass alle anderen Variablen nur Wahrheitswerte speichern.
2. Das Programm selbst deklariert (in einer geeigneten Notation), dass `i` und `j` vom Typ int sind, und dass `b` und `c` vom Typ bool (oder auch int) sind.
3. Das Typsystem erschließt die Typen aus der Verwendung der Variablen. Da `j` als Operand eines `+` vorkommt, muss `j` den Typ int haben. Da der Variablen `i` das Ergebnis einer Addition zugewiesen wird, muss auch sie den Typ int haben. Für die Variablen `b` und `c` lässt sich nur bestimmen, dass beide den gleichen Typ haben müssen, nicht aber, ob dies nun int oder bool sein soll.

Im Folgenden gehen wir davon aus, dass ein Programm auch die Typen der verwendeten (globalen) Variablen deklariert.

Definition 60 (Typkontext). Ein *Typkontext* $\Gamma :: \mathsf{Var} \Rightarrow \mathbb{T}$ ordnet jeder Variablen einen Typ zu.

Ein Programm besteht also aus der auszuführenden Anweisung c und einem Typkontext Γ.

Beispiel 61. Für die Anweisung

$$c \equiv \texttt{while (x <= 50) do (if (y) then x := x * 2 else x := x + 5)}$$

ist $\Gamma \equiv [\mathsf{x} \mapsto \mathsf{int}, \mathsf{y} \mapsto \mathsf{bool}]$ ein „passender“ Typkontext. In diesem Kontext hat die Variable `x` den Typ int, `y` den Typ bool.

6.6.2 Ein Typsystem für While_T

Ein Typsystem wird wie eine Semantik durch ein Regelsystem für eine Relation $\_ \vdash \_ :: \_$ definiert. Dabei bedeutet $\Gamma \vdash e :: \tau$, dass im Kontext Γ der Ausdruck e den Typ τ hat.

Definition 62 (Typregeln für Ausdrücke). Die Typregeln für Ausdrücke Exp sind im Einzelnen:

$$\textsc{TNum}\colon \Gamma \vdash n :: \mathsf{int} \qquad \textsc{TVar}\colon \frac{\Gamma(x) = \tau}{\Gamma \vdash x :: \tau} \qquad \textsc{TTrue}\colon \Gamma \vdash \texttt{true} :: \mathsf{bool}$$

$$\textsc{TMinus}\colon \frac{\Gamma \vdash e_1 :: \mathsf{int} \qquad \Gamma \vdash e_2 :: \mathsf{int}}{\Gamma \vdash e_1 \texttt{ - } e_2 :: \mathsf{int}} \qquad \textsc{TTimes}\colon \frac{\Gamma \vdash e_1 :: \mathsf{int} \qquad \Gamma \vdash e_2 :: \mathsf{int}}{\Gamma \vdash e_1 \texttt{ * } e_2 :: \mathsf{int}}$$

$$\textsc{TLeq}\colon \frac{\Gamma \vdash e_1 :: \mathsf{int} \quad \Gamma \vdash e_2 :: \mathsf{int}}{\Gamma \vdash e_1 \texttt{ <= } e_2 :: \mathsf{bool}} \qquad \textsc{TNot}\colon \frac{\Gamma \vdash e :: \mathsf{bool}}{\Gamma \vdash \texttt{not } e :: \mathsf{bool}} \qquad \textsc{TAnd}\colon \frac{\Gamma \vdash e_1 :: \mathsf{bool} \quad \Gamma \vdash e_2 :: \mathsf{bool}}{\Gamma \vdash e_1 \texttt{ \&\& } e_2 :: \mathsf{bool}}$$

Beispiel 63. Sei $\Gamma \equiv [\mathtt{x} \mapsto \mathsf{int}, \mathtt{y} \mapsto \mathsf{bool}]$. Dann gilt $\Gamma \vdash$ `(x <= 10) && (not y)` $:: \mathsf{bool}$, aber es gibt kein τ, für das $\Gamma \vdash$ `y <= x` $:: \tau$ gelte. Die Typaussage $\Gamma \vdash$ `(x <= 10) && (not y)` $:: \tau$ wird wie bei der Semantik durch einen Ableitungsbaum hergeleitet:

$$\dfrac{\dfrac{\dfrac{\Gamma(\mathtt{x}) = \mathsf{int}}{\Gamma \vdash \mathtt{x} :: \mathsf{int}}\,\textsc{TVar} \qquad \dfrac{}{\Gamma \vdash \texttt{10} :: \mathsf{int}}\,\textsc{TNum}}{\Gamma \vdash \texttt{x <= 10} :: \mathsf{bool}}\,\textsc{TLeq} \qquad \dfrac{\dfrac{\Gamma(\mathtt{y}) = \mathsf{bool}}{\Gamma \vdash \mathtt{y} :: \mathsf{bool}}\,\textsc{TVar}}{\Gamma \vdash \texttt{not y} :: \mathsf{bool}}\,\textsc{TNot}}{\Gamma \vdash \texttt{(x <= 10) \&\& (not y)} :: \mathsf{bool}}\,\textsc{TAnd}$$

Definition 64 (Typkorrektheit von Ausdrücken). Ein Ausdruck e heißt *typkorrekt* in einem Typkontext Γ, falls e einen Typ hat, d.h., falls es ein τ gibt, sodass $\Gamma \vdash e :: \tau$.

Definition 65 (Typkorrektheit von Anweisungen). Anweisungen selbst haben keinen Typ, müssen die Ausdrücke aber korrekt verwenden. Typkorrektheit $\Gamma \vdash c\ \surd$ für eine Anweisung c im Typkontext Γ ist auch durch ein Regelsystem definiert:

$$\textsc{TSkip}\colon \Gamma \vdash \texttt{skip}\ \surd \qquad \textsc{TAss}\colon \frac{\Gamma(x) = \tau \qquad \Gamma \vdash e :: \tau}{\Gamma \vdash x \texttt{ := } e\ \surd} \qquad \textsc{TSeq}\colon \frac{\Gamma \vdash c_1\ \surd \qquad \Gamma \vdash c_2\ \surd}{\Gamma \vdash c_1\texttt{; } c_2\ \surd}$$

$$\textsc{TIf}\colon \frac{\Gamma \vdash e :: \mathsf{bool} \qquad \Gamma \vdash c_1\ \surd \qquad \Gamma \vdash c_2\ \surd}{\Gamma \vdash \texttt{if (}e\texttt{) then } c_1 \texttt{ else } c_2\ \surd} \qquad \textsc{TWhile}\colon \frac{\Gamma \vdash e :: \mathsf{bool} \qquad \Gamma \vdash c\ \surd}{\Gamma \vdash \texttt{while (}e\texttt{) do } c\ \surd}$$

$$\textsc{TBlock}\colon \frac{\Gamma \vdash e :: \tau \qquad \Gamma[x \mapsto \tau] \vdash c\ \surd}{\Gamma \vdash \texttt{\{ var } x \texttt{ = } e\texttt{; } c \texttt{ \}}\ \surd}$$

Definition 66 (typisierbar, typkorrekte Programme). Eine Anweisung c heißt *typisierbar*, wenn es einen Kontext Γ mit $\Gamma \vdash c\ \surd$ gibt. Ein Programm $P \equiv (c, \Gamma)$ heißt *typkorrekt*, falls $\Gamma \vdash c\ \surd$.

Die Regel TBlock für Blöcke ist keine reine Typüberprüfung mehr, sondern inferiert den (neuen) Typ für x aus dem Typ des Initialisierungsausdrucks e. Wollte man eine reine Typüberprüfung, müsste ein Block entweder den neuen Typ für x deklarieren (z.B. `{ int x = 5; c }`) oder der Typ für x dürfte sich nicht ändern.

Bemerkung: Die Typsystem-Regeln für $\_ \vdash \_ :: \_$ und $\_ \vdash \_\ \surd$ spezifizieren bereits einen (terminierenden) Algorithmus, da die Annahmen der Regeln stets kleinere Ausdrücke oder Anweisungen enthalten als die Konklusion.

Übung: Welche der Programme vom Beginn des Abschnitts 6.6 sind typkorrekt – bei geeigneter Typdeklaration der globalen Variablen?

6.6.3 Small-Step-Semantik für While_T

Zustände müssen jetzt neben Zahlen auch Wahrheitswerte speichern können:

$$\Sigma \equiv \mathsf{Var} \Rightarrow \mathbb{Z} + \mathbb{B}$$

Bei der Auswertung eines Ausdrucks e in einem Zustand σ kann es nun passieren, dass die Werte in σ nicht zu den erwarteten Typen in e passen. Deswegen kann auch unsere neue Auswertungsfunktion $\mathcal{E}\,[\![e]\!]\,\sigma$ für den Ausdruck e im Zustand σ nur noch partiell sein:

$$\mathcal{E}\,[\![\_]\!]\,\_ :: \mathsf{Exp} \Rightarrow \Sigma \rightharpoonup \mathbb{Z} + \mathbb{B}$$

Wenn $\mathcal{E}\,[\![e]\!]\,\sigma$ nicht definiert ist, schreiben wir $\mathcal{E}\,[\![e]\!]\,\sigma = \bot$. Umgekehrt bedeutet $\mathcal{E}\,[\![e]\!]\,\sigma = v$ (wobei $v \in \mathbb{Z} + \mathbb{B}$), dass $\mathcal{E}\,[\![e]\!]\,\sigma$ definiert ist und den Wert v hat.

Definition 67 (Auswertungsfunktion für Ausdrücke). Die Auswertungsfunktion $\mathcal{E}\,[\![e]\!]\,\sigma$ kombiniert lediglich die bisherigen Auswertungsfunktionen $\mathcal{A}\,[\![\_]\!]\,\_$ und $\mathcal{B}\,[\![\_]\!]\,\_$:

$$\begin{array}{rcl|rcl}
\mathcal{E}\,[\![n]\!]\,\sigma & = & \mathcal{N}\,[\![n]\!] & \mathcal{E}\,[\![\texttt{true}]\!]\,\sigma & = & \mathbf{tt} \\
\mathcal{E}\,[\![x]\!]\,\sigma & = & \sigma(x) & \mathcal{E}\,[\![e_1\ \texttt{<=}\ e_2]\!]\,\sigma & = & \mathcal{E}\,[\![e_1]\!]\,\sigma \leq \mathcal{E}\,[\![e_2]\!]\,\sigma \\
\mathcal{E}\,[\![e_1\ \texttt{-}\ e_2]\!]\,\sigma & = & \mathcal{E}\,[\![e_1]\!]\,\sigma - \mathcal{E}\,[\![e_2]\!]\,\sigma & \mathcal{E}\,[\![\texttt{not}\ e]\!]\,\sigma & = & \neg\mathcal{E}\,[\![e]\!]\,\sigma \\
\mathcal{E}\,[\![e_1\ \texttt{*}\ e_2]\!]\,\sigma & = & \mathcal{E}\,[\![e_1]\!]\,\sigma \cdot \mathcal{E}\,[\![e_2]\!]\,\sigma & \mathcal{E}\,[\![e_1\ \texttt{\&\&}\ e_2]\!]\,\sigma & = & \mathcal{E}\,[\![e_1]\!]\,\sigma \wedge \mathcal{E}\,[\![e_2]\!]\,\sigma
\end{array}$$

Dabei sind die Operatoren $-$, $\cdot$, $\leq$, $\neg$ und $\wedge$ auch als partielle Funktionen auf dem Wertebereich $\mathbb{Z} + \mathbb{B}$ zu verstehen: Sie sind nur auf ihrem bisherigen Wertebereich ($\mathbb{Z}$ oder $\mathbb{B}$) definiert und für alle anderen Werte aus $\mathbb{Z} + \mathbb{B}$ undefiniert. Genauso sind sie undefiniert, wenn eines ihrer Argumente undefiniert ist.

Beispiel 68. $\mathcal{E}\,[\![\texttt{false \&\& (1 <= true)}]\!]\,\sigma = \bot$, weil $1 \leq \mathbf{tt} = \bot$ und damit auch $\mathbf{ff} \wedge \bot = \bot$.

Definition 69 (Small-Step-Semantik). Die Small-Step-Semantik für While_T besteht nun aus folgenden Regeln:

$$\textsc{Ass}^{\mathrm{T}}_{\mathrm{SS}}\colon \frac{\mathcal{E}\,[\![e]\!]\,\sigma = v}{\langle x\ \texttt{:=}\ e,\ \sigma\rangle \to_1 \langle \texttt{skip},\ \sigma[x \mapsto v]\rangle}$$

$$\textsc{Seq1}^{\mathrm{T}}_{\mathrm{SS}}\colon \frac{\langle c_0,\ \sigma\rangle \to_1 \langle c_0',\ \sigma'\rangle}{\langle c_0\texttt{;}\ c_1,\ \sigma\rangle \to_1 \langle c_0'\texttt{;}\ c_1,\ \sigma'\rangle} \qquad \textsc{Seq2}^{\mathrm{T}}_{\mathrm{SS}}\colon \langle \texttt{skip;}\ c,\ \sigma\rangle \to_1 \langle c,\ \sigma\rangle$$

$$\textsc{IfTT}^{\mathrm{T}}_{\mathrm{SS}}\colon \frac{\mathcal{E}\,[\![e]\!]\,\sigma = \mathbf{tt}}{\langle \texttt{if (}e\texttt{) then}\ c_0\ \texttt{else}\ c_1,\ \sigma\rangle \to_1 \langle c_0,\ \sigma\rangle}$$

$$\textsc{IfFF}^{\mathrm{T}}_{\mathrm{SS}}\colon \frac{\mathcal{E}\,[\![e]\!]\,\sigma = \mathbf{ff}}{\langle \texttt{if (}e\texttt{) then}\ c_0\ \texttt{else}\ c_1,\ \sigma\rangle \to_1 \langle c_1,\ \sigma\rangle}$$

$$\textsc{While}^{\mathrm{T}}_{\mathrm{SS}}\colon \langle \texttt{while (}e\texttt{) do}\ c,\ \sigma\rangle \to_1 \langle \texttt{if (}e\texttt{) then}\ c\texttt{; while (}e\texttt{) do}\ c\ \texttt{else skip},\ \sigma\rangle$$

$$\textsc{Block1}^{\mathrm{T}}_{SS}\colon \frac{\mathcal{E}\,[\![e]\!]\,\sigma = v \qquad \langle c,\ \sigma[x \mapsto v]\rangle \to_1 \langle c',\ \sigma'\rangle}{\langle \texttt{\{ var}\ x\ \texttt{=}\ e\texttt{;}\ c\ \texttt{\}},\ \sigma\rangle \to_1 \langle \texttt{\{ var}\ x\ \texttt{=}\ \mathcal{V}^{-1}\,[\![\sigma'(x)]\!]\texttt{;}\ c'\ \texttt{\}},\ \sigma'[x \mapsto \sigma(x)]\rangle}$$

$$\textsc{Block2}^{\mathrm{T}}_{SS}\colon \frac{\mathcal{E}\,[\![e]\!]\,\sigma \neq \bot}{\langle \texttt{\{ var}\ x\ \texttt{=}\ e\texttt{; skip \}},\ \sigma\rangle \to_1 \langle \texttt{skip},\ \sigma\rangle}$$

Dabei ist $\mathcal{V}^{-1}[\![v]\!]$ die Umkehrung von $\mathcal{E}[\![\_]\!]\_$ für Werte $v \in \mathbb{Z} + \mathbb{B}$, also die Verallgemeinerung von $\mathcal{N}^{-1}[\![\_]\!]$ auf den neuen Wertebereich für Variablen. Regel $\textsc{Block2}^{\mathrm{T}}_{SS}$ ist nur anwendbar, wenn der Initialisierungsausdruck e keine Typfehler bei der Auswertung hervorruft.

Übung: Zeigen Sie, dass es neben $\langle \texttt{skip},\ \sigma\rangle$ weitere blockierte Konfigurationen in dieser Small-Step-Semantik gibt. Woran liegt das?

6.6.4 Typsicherheit von While$_T$

Die Semantik ist unabhängig vom Typsystem und seinen Regeln. Semantik beschreibt das *dynamische* Verhalten eines Programms, das Typsystem eine *statische* Programmanalyse. Wie alle Programmanalysen gibt ein Typsystem ein Korrektheitsversprechen, das mittels der Semantik bewiesen werden kann.

Definition 70 (Korrektheit, Typsicherheit, Vollständigkeit). Ein Typsystem ist *korrekt*, falls es zur Laufzeit keine Typfehler gibt. Laufzeit-Typfehler äußern sich in blockierten Konfigurationen der Small-Step-Semantik, die keine Endzustände (`skip`) sind. Sprachen mit einem korrekten Typsystem heißen *typsicher*. Dual zur Korrektheit ist der Begriff der Vollständigkeit: Das Typsystem ist *vollständig*, wenn jedes Programm ohne Laufzeittypfehler typkorrekt ist.

Korrektheit ist wünschenswert, Vollständigkeit im Allgemeinen (für korrekte Typsysteme) unerreichbar, weil dann das Typsystem nicht mehr entscheidbar sein kann. Viele moderne Programmiersprachen sind (nachgewiesenermaßen) typsicher: ML, Java, C#, Haskell. Nicht typsicher sind beispielsweise Eiffel, C, C++ und Pascal.

Beispiel 71. Das Typsystem für While$_T$ ist nicht vollständig. Das Programm `x := 0; x := true` ist nicht typkorrekt, verursacht aber trotzdem keine Laufzeittypfehler.

Um die Typsicherheit von While$_T$ formulieren zu können, brauchen wir noch zwei Begriffe: Den Typ eines Wertes und Zustandskonformanz.

Definition 72 (Typ eines Wertes). Der *Typ* $\mathrm{type}(v)$ eines Wertes $v \in \mathbb{Z} + \mathbb{B}$ ist:

$$\mathrm{type}(v) \equiv \begin{cases} \mathsf{int} & \text{falls } v \in \mathbb{Z} \\ \mathsf{bool} & \text{falls } v \in \mathbb{B} \end{cases}$$

Definition 73 (Zustandskonformanz). Ein Zustand σ ist zu einem Typkontext Γ *konformant*, notiert als $\sigma :: \Gamma$, wenn $\mathrm{type}(\sigma(x)) = \Gamma(x)$ für alle $x \in \mathsf{Var}$.

Typsicherheit für eine Small-Step-Semantik besteht klassischerweise aus zwei Teilen: Fortschritt und Typerhaltung (progress und preservation) nach Wright und Felleisen.

Theorem 74 (Typsicherheit). Sei $\Gamma \vdash c\ \surd$ und $\sigma :: \Gamma$.
Progress Wenn $c \neq \texttt{skip}$, dann gibt es c' und σ' mit $\langle c,\ \sigma\rangle \to_1 \langle c',\ \sigma'\rangle$.
Preservation Wenn $\langle c,\ \sigma\rangle \to_1 \langle c',\ \sigma'\rangle$, dann $\Gamma \vdash c'\ \surd$ und $\sigma' :: \Gamma$.

Den Beweis teilen wir auf die folgenden Hilfslemmata auf:

Lemma 75 (Typkorrektheit von $\mathcal{E}[\![\_]\!]\_$).
Wenn $\Gamma \vdash e :: \tau$ und $\sigma :: \Gamma$, dann ist $\mathcal{E}[\![e]\!]\sigma$ definiert und $\mathrm{type}(\mathcal{E}[\![e]\!]\sigma) = \tau$.

Beweis. Regel-Induktion über $\Gamma \vdash e :: \tau$.

- Fälle TNum, TTrue: Trivial.
- Fall TVar: Definiertheit ist trivial. Wegen Zustandskonformanz $\sigma :: \Gamma$ gilt: $\text{type}(\mathcal{E}\,[\![e]\!]\,\sigma) = \text{type}(\sigma(x)) = \Gamma(x) = \tau$.
- Fall TMinus:
  Induktionsannahmen: $\mathcal{E}\,[\![e_1]\!]\,\sigma$ und $\mathcal{E}\,[\![e_2]\!]\,\sigma$ sind definiert mit $\text{type}(\mathcal{E}\,[\![e_1]\!]\,\sigma) = \text{type}(\mathcal{E}\,[\![e_2]\!]\,\sigma) = \mathsf{int}$. Nach Definition von type(_) ist somit $\mathcal{E}\,[\![e_1]\!]\,\sigma \in \mathbb{Z}$ und $\mathcal{E}\,[\![e_1]\!]\,\sigma \in \mathbb{Z}$. Also ist auch $\mathcal{E}\,[\![e_1 \texttt{ - } e_2]\!]\,\sigma$ definiert mit $\mathcal{E}\,[\![e_1 \texttt{ - } e_2]\!]\,\sigma \in \mathbb{Z}$, also $\text{type}(\mathcal{E}\,[\![e_1 \texttt{ - } e_2]\!]\,\sigma) = \mathsf{int}$.
- Fälle TTimes, TLeq, TNot, TAnd: Analog. □

Lemma 76 (Fortschritt).
Wenn $\Gamma \vdash c\ \surd$ und $\sigma :: \Gamma$ mit $c \neq \texttt{skip}$, dann gibt es c' und σ' mit $\langle c,\ \sigma\rangle \to_1 \langle c',\ \sigma'\rangle$.

Beweis. Induktion über $\Gamma \vdash c\ \surd$ oder c direkt – analog zu Lem. 16 (σ beliebig). Für die Definiertheit von $\mathcal{E}\,[\![e]\!]\,\sigma$ in den Regelannahmen verwendet man Lem. 75. Bei Blöcken ist wie bei der Sequenz eine Fallunterscheidung über $c = \texttt{skip}$ nötig, für den induktiven Fall $c \neq \texttt{skip}$ braucht man die Induktionshypothese mit $\sigma[x \mapsto v]$ für σ. □

Lemma 77 (Erhalt der Zustandskonformanz).
Wenn $\Gamma \vdash c\ \surd$, $\sigma :: \Gamma$ und $\langle c,\ \sigma\rangle \to_1 \langle c',\ \sigma'\rangle$, dann $\sigma' :: \Gamma$.

Beweis. Regel-Induktion über $\langle c,\ \sigma\rangle \to_1 \langle c',\ \sigma'\rangle$ (Γ beliebig).

- Fall $\text{Ass}^{\text{T}}_{\text{SS}}$: Zu zeigen: Wenn $\Gamma \vdash x \texttt{ := } e\ \surd$, $\sigma :: \Gamma$ und $\mathcal{E}\,[\![e]\!]\,\sigma = v$, dann $\sigma[x \mapsto v] :: \Gamma$.

  Aus $\Gamma \vdash x \texttt{ := } e\ \surd$ erhält man mit Regelinversion (TAss) τ mit $\Gamma(x) = \tau$ und $\Gamma \vdash e :: \tau$. Aus $\Gamma \vdash e :: \tau$ und $\sigma :: \Gamma$ folgt nach Lem. 75, dass $\text{type}(v) = \tau$. Zusammen mit $\sigma :: \Gamma$ und $\Gamma(x) = \tau$ folgt die Behauptung $\sigma[x \mapsto v] :: \Gamma$.
- Fall $\text{Seq1}^{\text{T}}_{\text{SS}}$: Fall-Annahmen: $\langle c_1,\ \sigma\rangle \to_1 \langle c_1',\ \sigma'\rangle$, $\Gamma \vdash c_1\texttt{ ; }\ c_2\ \surd$, $\sigma :: \Gamma$.
  Induktionsannahme: Für alle Γ gilt: Wenn $\Gamma \vdash c_1\ \surd$ und $\sigma :: \Gamma$, dann $\sigma' :: \Gamma$.
  Zu zeigen: $\sigma' :: \Gamma$.

  Aus $\Gamma \vdash c_1\texttt{ ; }\ c_2\ \surd$ erhält man durch Regelinversion (TSeq), dass $\Gamma \vdash c_1\ \surd$ und $\Gamma \vdash c_2\ \surd$. Mit $\sigma :: \Gamma$ folgt die Behauptung aus der Induktionsannahme.
- Fälle $\text{Seq2}^{\text{T}}_{\text{SS}}$, $\text{IfTT}^{\text{T}}_{\text{SS}}$, $\text{IfFF}^{\text{T}}_{\text{SS}}$, $\text{While}^{\text{T}}_{\text{SS}}$, $\text{Block2}^{\text{T}}_{SS}$: Trivial, da $\sigma' = \sigma$.
- Fall $\text{Block1}^{\text{T}}_{SS}$:
  Fall-Annahmen: $\mathcal{E}\,[\![e]\!]\,\sigma = v$, $\langle c,\ \sigma[x \mapsto v]\rangle \to_1 \langle c',\ \sigma'\rangle$, $\Gamma \vdash \texttt{\{ var } x \texttt{ = } e\texttt{; }\ c\texttt{ \}}\ \surd$ und $\sigma :: \Gamma$.
  Induktionsannahme: Für beliebige Γ gilt: Wenn $\Gamma \vdash c\ \surd$ und $\sigma[x \mapsto v] :: \Gamma$, dann $\sigma' :: \Gamma$.
  Zu zeigen: $\sigma'[x \mapsto \sigma(x)] :: \Gamma$.

  Aus $\Gamma \vdash \texttt{\{ var } x \texttt{ = } e\texttt{; }\ c\texttt{ \}}\ \surd$ erhält man durch Regelinversion (TBlock) τ mit $\Gamma \vdash e :: \tau$ und $\Gamma[x \mapsto \tau] \vdash c\ \surd$. Aus $\mathcal{E}\,[\![e]\!]\,\sigma = v$, $\Gamma \vdash e :: \tau$ und $\sigma :: \Gamma$ folgt mit Lem. 75, dass $\text{type}(v) = \tau$. Zusammen mit $\sigma :: \Gamma$ gilt damit auch $\sigma[x \mapsto v] :: \Gamma[x \mapsto \tau]$. Aus der Induktionsannahme mit $\Gamma[x \mapsto \tau]$ für Γ erhält man damit $\sigma' :: \Gamma[x \mapsto \tau]$. Wegen $\sigma :: \Gamma$ ist $\Gamma(x) = \text{type}(\sigma(x))$. Damit gilt auch die Behauptung $\sigma'[x \mapsto \sigma(x)] :: \Gamma$, da $\Gamma[x \mapsto \tau][x \mapsto \text{type}(\sigma(x))] = \Gamma$. □

Lemma 78 (Subject reduction). Wenn $\langle c,\ \sigma\rangle \to_1 \langle c',\ \sigma'\rangle$, $\Gamma \vdash c\ \surd$ und $\sigma :: \Gamma$, dann $\Gamma \vdash c'\ \surd$.

Beweis. Regel-Induktion über $\langle c,\ \sigma\rangle \to_1 \langle c',\ \sigma'\rangle$ (Γ beliebig).

- Fälle $\text{Ass}^{\text{T}}_{\text{SS}}$, $\text{Block2}^{\text{T}}_{SS}$: Trivial mit Regel TSkip.
- Fall $\text{Seq1}^{\text{T}}_{\text{SS}}$:
  Induktionsannahme: Wenn $\Gamma \vdash c_1\ \surd$ und $\sigma :: \Gamma$, dann $\Gamma \vdash c_1'\ \surd$.
  Zu zeigen: Wenn $\Gamma \vdash c_1\texttt{ ; }\ c_2\ \surd$ und $\sigma :: \Gamma$, dann $\Gamma \vdash c_1'\texttt{ ; }\ c_2\ \surd$.

Aus $\Gamma \vdash c_1\texttt{;}\ c_2\ \surd$ erhält man durch Regelinversion (TSeq), dass $\Gamma \vdash c_1\ \surd$ und $\Gamma \vdash c_2\ \surd$. Mit $\sigma :: \Gamma$ folgt aus der Induktionsannahme, dass $\Gamma \vdash c_1'\ \surd$. Zusammen mit $\Gamma \vdash c_2\ \surd$ folgt die Behauptung mit Regel TSeq.

- Fälle $\text{Seq2}^{\text{T}}_{\text{SS}}$, $\text{IfTT}^{\text{T}}_{\text{SS}}$, $\text{IfFF}^{\text{T}}_{\text{SS}}$: Trivial mit Regelinversion (TSeq bzw. TIf).
- Fall $\text{While}^{\text{T}}_{\text{SS}}$: Aus $\Gamma \vdash \texttt{while (}e\texttt{) do }c\ \surd$ folgt mit Regelinversion, dass $\Gamma \vdash e :: \mathsf{bool}$ und $\Gamma \vdash c\ \surd$. Damit gilt:

$$\dfrac{\Gamma \vdash e :: \mathsf{bool} \qquad \dfrac{\dfrac{\Gamma \vdash c\ \surd \qquad \Gamma \vdash \texttt{while (}e\texttt{) do }c\ \surd}{\Gamma \vdash c\texttt{; while (}e\texttt{) do }c\ \surd}\ \text{TSeq} \qquad \dfrac{}{\Gamma \vdash \texttt{skip}\ \surd}\ \text{TSkip}}{\Gamma \vdash \texttt{if (}e\texttt{) then }c\texttt{; while (}e\texttt{) do }c\texttt{ else skip}\ \surd}\ \text{TIf}$$

- Fall $\text{Block1}^{\text{T}}_{SS}$:
  Fall-Annahmen: $\langle c,\ \sigma[x \mapsto v]\rangle \to_1 \langle c',\ \sigma'\rangle$, $\Gamma \vdash \texttt{\{ var }x\texttt{ = }e\texttt{;}\ c\texttt{ \}}\ \surd$, $\sigma :: \Gamma$ und $\mathcal{E}\,[\![e]\!]\,\sigma = v$.
  Induktionsannahme: Für beliebige Γ gilt: Wenn $\Gamma \vdash c\ \surd$ und $\sigma[x \mapsto v] :: \Gamma$, dann $\Gamma \vdash c'\ \surd$.
  Zu zeigen: $\Gamma \vdash \texttt{\{ var }x\texttt{ = }\mathcal{V}^{-1}\,[\![\sigma'(x)]\!]\texttt{;}\ c'\texttt{ \}}\ \surd$.

  Aus $\Gamma \vdash \texttt{\{ var }x\texttt{ = }e\texttt{;}\ c\texttt{ \}}\ \surd$ erhält man durch Regelinversion (TBlock) ein τ mit $\Gamma \vdash e :: \tau$ und $\Gamma[x \mapsto \tau] \vdash c\ \surd$. Mit $\mathcal{E}\,[\![e]\!]\,\sigma = v$ und $\sigma :: \Gamma$ ist wieder $\text{type}(v) = \tau$ nach Lem. 75, also auch $\sigma[x \mapsto v] :: \Gamma[x \mapsto \tau]$, und damit folgt aus der Induktionsannahme mit $\Gamma[x \mapsto \tau]$ für Γ, dass $\Gamma[x \mapsto \tau] \vdash c'\ \surd$.

  Mit Lem. 77 gilt $\sigma' :: \Gamma[x \mapsto \tau]$. Damit gilt $\text{type}(\sigma'(x)) = \tau$ und somit $\Gamma \vdash \mathcal{V}^{-1}\,[\![\sigma'(x)]\!] :: \tau$ nach Regeln TNum bzw. TTrue und TNot. Zusammen mit $\Gamma[x \mapsto \tau] \vdash c'\ \surd$ folgt die Behauptung nach Regel TBlock. □

Lem. 76, 77 und 78 zusammen beweisen das Typsicherheitstheorem 74. Folgendes Korollar über die maximalen Ableitungssequenzen ist eine unmittelbare Folgerung daraus:

Korollar 79 (Vollständige Auswertung).
Wenn $\Gamma \vdash c\ \surd$ und $\sigma :: \Gamma$, dann gibt es entweder ein σ' mit $\langle c,\ \sigma\rangle \overset{*}{\to}_1 \langle\texttt{skip},\ \sigma'\rangle$ oder $\langle c,\ \sigma\rangle \overset{\infty}{\to}_1$.

Beweis. Angenommen, $\langle c,\ \sigma\rangle \overset{\infty}{\not\to}_1$. Dann gibt es ein c' und σ' mit $\langle c,\ \sigma\rangle \overset{*}{\to}_1 \langle c',\ \sigma'\rangle$, so dass $\langle c',\ \sigma'\rangle$ blockiert. Mittels Induktion über die transitive Hülle erhält man aus Lem. 78 und Lem. 77, dass $\Gamma \vdash c'\ \surd$ und $\sigma' :: \Gamma$. Nach dem Fortschrittslemma 76 ist $\langle c',\ \sigma'\rangle$ aber nur dann blockiert, wenn $c' = \texttt{skip}$, was zu zeigen war. □

7 Denotationale Semantik

Eine denotationale Semantik kümmert sich nur um den Effekt einer Programmausführung. Im Gegensatz dazu haben sich die operationalen Semantiken auch explizit um die Zwischenzustände gekümmert: Am deutlichsten ist dies bei der Small-Step-Semantik, deren Ableitungsfolgen erst einmal alle Zwischenschritte enthalten – das interessante Gesamtverhalten erhält man erst durch die Abstraktion zur transitiven Hülle.

Nun könnte man einfach weiter abstrahieren: Da die operationalen Semantiken für While deterministisch sind (vgl. Thm. 10 und Kor. 18), kann man diese Relationen auch als (partielle) Funktionen $\mathcal{D}[\![c]\!]$ auf Zuständen auffassen: Nimmt man die Big-Step-Semantik, erhält man:

$$\mathcal{D}[\![c]\!]\,\sigma = \begin{cases} \sigma' & \text{falls } \langle c,\ \sigma\rangle \Downarrow \sigma' \\ \bot & \text{falls } \nexists \sigma'.\ \langle c,\ \sigma\rangle \Downarrow \sigma' \end{cases}$$

Auch auf Basis der Small-Step-Semantik ließe sich diese Funktion $\mathcal{D}[\![c]\!]$ für das Programm c entsprechend definieren:

$$\mathcal{D}[\![c]\!]\,\sigma = \begin{cases} \sigma' & \text{falls } \langle c,\ \sigma\rangle \overset{*}{\to}_1 \langle \texttt{skip},\ \sigma'\rangle \\ \bot & \text{falls } \langle c,\ \sigma\rangle \overset{\infty}{\to}_1 \end{cases}$$

In beiden Varianten gewinnt man so allerdings nichts, da Beweise von Aussagen über $\mathcal{D}[\![c]\!]$ erst einmal die Definition auffalten müssen und man dann wieder bei den Ausführungsdetails der operationalen Semantik landet. Der große Vorteil einer denotationellen Semantik liegt allerdings genau darin, dass man viele Theoreme *ohne* Rückgriff auf Ableitungsbäume zeigen und verwenden kann.

Eine denotationale Semantik sollte *kompositional* sein, d.h., je eine Funktion für die syntaktischen Kategorien (Aexp, Bexp und Com) *rekursiv über dem Syntaxbaum* definieren. Dadurch ordnet sie jedem Syntaxelement (Syntaxbaum) ein mathematisches Objekt, i.d.R. eine Funktion, zu, das den Effekt bzw. das Ergebnis dieses syntaktischen Konstrukts beschreibt.

Beispiel 80. Die Auswertungsfunktionen $\mathcal{A}[\![\_]\!]$ bzw. $\mathcal{B}[\![\_]\!]$ für arithmetische bzw. boolesche Ausdrücke sind bereits so definiert. Eigentlich sind dies bereits denotationale Semantiken und keine operationalen. Hier nochmals die Definition von $\mathcal{A}[\![\_]\!]$ vom Typ $\mathsf{Aexp} \Rightarrow (\Sigma \Rightarrow \mathbb{Z})$:

$$\begin{array}{rcl} \mathcal{A}[\![n]\!]\,\sigma & = & \mathcal{N}[\![n]\!] \\ \mathcal{A}[\![x]\!]\,\sigma & = & \sigma(x) \\ \mathcal{A}[\![a_1\ \texttt{-}\ a_2]\!]\,\sigma & = & \mathcal{A}[\![a_1]\!]\,\sigma - \mathcal{A}[\![a_2]\!]\,\sigma \\ \mathcal{A}[\![a_1\ \texttt{*}\ a_2]\!]\,\sigma & = & \mathcal{A}[\![a_1]\!]\,\sigma \cdot \mathcal{A}[\![a_2]\!]\,\sigma \end{array}$$

Übung: Finden Sie eine operationale Semantik für arithmetische Ausdrücke, sowohl Big-Step als auch Small-Step.

7.1 Denotationale Semantik

Bei unserer imperativen While-Sprache ist das Ergebnis einer Ausführung eines Programms c die Zustandsänderung. Es bietet sich an, diese Änderung als Funktion von Anfangs- zu Endzustand zu beschreiben. Da ein Programm bei manchen Anfangszuständen möglicherweise nicht terminiert, müssen diese Funktionen partiell sein. Die mathematischen Bedeutungsobjekte für Com sind damit partielle Funktionen auf Zuständen:

$$\mathcal{D}[\![\_]\!] :: \mathsf{Com} \Rightarrow (\Sigma \rightharpoonup \Sigma)$$

Die obige Definition von $\mathcal{D}[\![\_]\!]$ über die Big-Step-Semantik entspricht genau diesem Typ. Sie ist aber nicht kompositional, selbst wenn man die Definition der Big-Step-Semantik einsetzt: Für `while (`b`) do` c ergibt sich aus den Regeln $\textsc{WhileFF}_{\text{BS}}$ und $\textsc{WhileTT}_{\text{BS}}$:

$$\mathcal{D}[\![\texttt{while (}b\texttt{) do } c]\!](\sigma) = \begin{cases} \sigma & \text{falls } \mathcal{B}[\![b]\!]\sigma = \mathbf{ff} \\ \mathcal{D}[\![\texttt{while (}b\texttt{) do } c]\!](\mathcal{D}[\![c]\!](\sigma)) & \text{falls } \mathcal{B}[\![b]\!]\sigma = \mathbf{tt} \end{cases}$$

Dies ist aber keine kompositionale Definition, weil der zu definierende Term $\mathcal{D}[\![\texttt{while (}b\texttt{) do } c]\!]$ sowohl links wie auch rechts des Gleichheitszeichens in gleicher Form auftritt. Genau genommen ist dies erst einmal überhaupt keine Definition, weil nicht klar ist, dass es eine Funktion mit dieser Eigenschaft überhaupt gibt und dass diese Eigenschaft die Funktion eindeutig festlegt.

Der Standard-Ansatz, um solche rekursiven Gleichungen in eine Definition zu verwandeln, ist, das zu Definierende in einen Parameter eines Funktionals umzuwandeln und es als einen ausgewählten Fixpunkt des Funktionals zu definieren. Damit ergibt sich folgendes Funktional $F :: (\Sigma \rightharpoonup \Sigma) \Rightarrow \Sigma \rightharpoonup \Sigma$ aus der rekursiven Gleichung für $\mathcal{D}[\![\texttt{while (}b\texttt{) do } c]\!]$:

$$F(f)(\sigma) = \begin{cases} \sigma & \text{falls } \mathcal{B}[\![b]\!]\sigma = \mathbf{ff} \\ f(\mathcal{D}[\![c]\!]\sigma) & \text{falls } \mathcal{B}[\![b]\!]\sigma = \mathbf{tt} \end{cases}$$

Beispiel 81. Für das Programm `while (not (x == 0)) do x := x - 1` vereinfacht sich dieses Funktional zu:

$$F(f)(\sigma) = \begin{cases} \sigma & \text{falls } \sigma(\mathtt{x}) = 0 \\ f(\sigma[\mathtt{x} \mapsto \sigma(\mathtt{x}) - 1]) & \text{falls } \sigma(\mathtt{x}) \neq 0 \end{cases}$$

Ein Fixpunkt f von F erfüllt $F(f) = f$. Damit ergeben sich folgende Anforderungen an alle möglichen Lösungen f:

- f muss alle Zustände σ mit $\sigma(\mathtt{x}) = 0$ auf σ selbst abbilden.
- f muss alle Zustände σ mit $\sigma(\mathtt{x}) \neq 0$ auf den gleichen Zustand abbilden wie $\sigma[\mathtt{x} \mapsto \sigma(\mathtt{x}) - 1]$.

Sehr viele Funktionen erfüllen diese Kriterien, z.B. alle der Form

$$f(\sigma) = \begin{cases} \sigma[\mathtt{x} \mapsto 0] & \text{falls } \sigma(\mathtt{x}) \geq 0 \\ \sigma' & \text{falls } \sigma(\mathtt{x}) < 0 \end{cases}$$

wobei $\sigma' \in \Sigma$ beliebig ist. Ebenso ist

$$f^*(\sigma) = \begin{cases} \sigma[\mathtt{x} \mapsto 0] & \text{falls } \sigma(\mathtt{x}) \geq 0 \\ \bot & \text{falls } \sigma(\mathtt{x}) < 0 \end{cases}$$

ein Fixpunkt von F. All diese Fixpunkte unterscheiden sich nur an den Zuständen, für die die Ausführung des Programms nicht terminiert.

Beispiel 82. Es gibt auch Funktionale, die gar keinen Fixpunkt haben:

$$F(f) = \begin{cases} f_1 & \text{falls } f = f_2 \\ f_2 & \text{sonst} \end{cases}$$

hat für $f_1 \neq f_2$ keinen Fixpunkt.

Die Lösung des Problems mit der Existenz und Eindeutigkeit einer Lösung der rekursiven Spezifikation wird sein:

1. eine Menge von Funktionalen des Typs $(\Sigma \rightharpoonup \Sigma) \Rightarrow (\Sigma \rightharpoonup \Sigma)$ zu finden, die immer einen Fixpunkt haben, und dann den „undefiniertesten“ Fixpunkt auswählen (Kap. 7.2);
2. zu zeigen, dass alle Funktionale, die durch While-Programme entstehen, in dieser Menge enthalten sind (Kap. 7.3).

Definition 83 (Fixpunktiteration). Eine Funktion $f :: D \Rightarrow D$ kann wie folgt *iteriert* werden:

$$f^0(d) = d \qquad f^{n+1}(d) = f(f^n(d))$$

Man kann nun ein Funktional iterieren und erhält dadurch schrittweise eine Annäherung an die Lösung der Gleichung, für die das Funktional steht. Dabei startet man mit der überall undefinierten Funktion $\bot$ (für alle σ gilt $\bot(\sigma) = \bot$), die keinerlei Information trägt.

Beispiel 84. Für das Funktional F aus Beispiel 81 ergibt sich folgende Fixpunkt-Iteration:

$$F^0(\bot) = \bot$$

$$F^1(\bot)(\sigma) = F(F^0(\bot))(\sigma) = F(\bot)(\sigma) = \begin{cases} \sigma & \text{falls } \sigma(\mathtt{x}) = 0 \\ \bot(\sigma[\mathtt{x} \mapsto \sigma(\mathtt{x}) - 1) = \bot & \text{falls } \sigma(\mathtt{x}) \neq 0 \end{cases}$$

$$F^2(\bot)(\sigma) = F(F^1(\bot))(\sigma) = \begin{cases} \sigma & \text{falls } \sigma(\mathtt{x}) = 0 \\ F^1(\bot)(\sigma[\mathtt{x} \mapsto \sigma(\mathtt{x}) - 1]) & \text{falls } \sigma(\mathtt{x}) \neq 0 \end{cases}$$

$$= \begin{cases} \sigma & \text{falls } \sigma(\mathtt{x}) = 0 \\ \sigma[\mathtt{x} \mapsto \sigma(\mathtt{x}) - 1] & \text{falls } \sigma[\mathtt{x} \mapsto \sigma(\mathtt{x}) - 1](\mathtt{x}) = 0 \text{ und } \sigma(\mathtt{x}) \neq 0 \\ \bot & \text{falls } \sigma[\mathtt{x} \mapsto \sigma(\mathtt{x}) - 1](\mathtt{x}) \neq 0 \text{ und } \sigma(\mathtt{x}) \neq 0 \end{cases}$$

$$= \begin{cases} \sigma & \text{falls } \sigma(\mathtt{x}) = 0 \\ \sigma[\mathtt{x} \mapsto 0] & \text{falls } \sigma(\mathtt{x}) = 1 \\ \bot & \text{sonst} \end{cases}$$

$$F^3(\bot)(\sigma) = F(F^2(\bot))(\sigma) = \begin{cases} \sigma & \text{falls } \sigma(\mathtt{x}) = 0 \\ F^2(\bot)(\sigma[\mathtt{x} \mapsto \sigma(\mathtt{x}) - 1]) & \text{falls } \sigma(\mathtt{x}) \neq 0 \end{cases}$$

$$= \begin{cases} \sigma & \text{falls } \sigma(\mathtt{x}) = 0 \\ \sigma[\mathtt{x} \mapsto \sigma(\mathtt{x}) - 1] & \text{falls } \sigma[\mathtt{x} \mapsto \sigma(\mathtt{x}) - 1](\mathtt{x}) = 0 \text{ und } \sigma(\mathtt{x}) \neq 0 \\ \sigma[\mathtt{x} \mapsto \sigma(\mathtt{x}) - 1,\ \mathtt{x} \mapsto 0] & \text{falls } \sigma[\mathtt{x} \mapsto \sigma(\mathtt{x}) - 1](\mathtt{x}) = 1 \text{ und } \sigma(\mathtt{x}) \neq 0 \\ \bot & \text{sonst} \end{cases}$$

$$= \begin{cases} \sigma & \text{falls } \sigma(\mathtt{x}) = 0 \\ \sigma[\mathtt{x} \mapsto 0] & \text{falls } \sigma(\mathtt{x}) \in \{\, 1,\ 2 \,\} \\ \bot & \text{sonst} \end{cases}$$

Allgemein gilt:

$$F^{n+1}(\bot)(\sigma) = \begin{cases} \sigma & \text{falls } \sigma(\mathtt{x}) = 0 \\ \sigma[\mathtt{x} \mapsto 0] & \text{falls } \sigma(\mathtt{x}) \in \{\, 1, \ldots, n \,\} \\ \bot & \text{sonst} \end{cases}$$

Die Glieder der Folge $F^n(\bot)$ sind mit wachsendem n an immer mehr Stellen definiert. Keines der Elemente der Folge ist selbst ein Fixpunkt, erst der „Grenzwert“ der Folge für $n \to \infty$, im Beispiel 81 die Funktion f^*.

Beispiel 85. Die Fixpunktiteration tritt auch in der Analysis auf, beispielsweise beim Newton-Verfahren zur Nullstellenberechnung für eine stetig differenzierbare Funktion $f :: \mathbb{R} \Rightarrow \mathbb{R}$. Dazu bildet man das Funktional $F :: \mathbb{R} \Rightarrow \mathbb{R}$ mit

$$F(x) = x - \frac{f(x)}{f'(x)}$$

Für alle Nullstellen x^* von f, an denen die Ableitung nicht verschwindet ($f'(x^*) \neq 0$), gilt:

$$F(x^*) = x^* - \frac{f(x^*)}{f'(x^*)} = x^* - \frac{0}{f'(x^*)} = x^*$$

d.h., x^* ist ein Fixpunkt von F. Um nun eine Nullstelle anzunähern, berechnet man die Folge $F^n(x_0)$ für einen (geeigneten) Startwert x_0. Der gesuchte Fixpunkt ist dann der (analytische) Grenzwert dieser Folge, falls sie konvergiert.

Kehren wir nun zur Definition von $\mathcal{D}[\![\_]\!]$ zurück. Nehmen wir vorläufig an, dass wir einen Fixpunktoperator FIX :: $((\Sigma \rightharpoonup \Sigma) \Rightarrow (\Sigma \rightharpoonup \Sigma)) \Rightarrow (\Sigma \rightharpoonup \Sigma)$ zur Verfügung haben, der den Grenzwert der Fixpunktiteration des übergebenen Funktionals berechnet. Im folgenden Abschnitt über Fixpunkttheorie werden wir sehen, dass es einen solchen für unsere Zwecke ausreichenden Operator gibt.

Definition 86 (Denotationale Semantik).
Die denotationale Semantik $\mathcal{D}[\![\_]\!]$ für While ist definiert durch

$$\begin{aligned}
\mathcal{D}[\![\texttt{skip}]\!] &= id \\
\mathcal{D}[\![x \texttt{ := } a]\!] &= \lambda\sigma.\ \sigma[x \mapsto \mathcal{A}[\![a]\!]\,\sigma] \\
\mathcal{D}[\![c_1 \texttt{;}\ c_2]\!] &= \mathcal{D}[\![c_2]\!] \circ \mathcal{D}[\![c_1]\!] \\
\mathcal{D}[\![\texttt{if (}b\texttt{) then } c_1 \texttt{ else } c_2]\!] &= \mathrm{IF}\,(\mathcal{B}[\![b]\!],\ \mathcal{D}[\![c_1]\!],\ \mathcal{D}[\![c_2]\!]) \\
\mathcal{D}[\![\texttt{while (}b\texttt{) do } c]\!] &= \mathrm{FIX}\,(\lambda f.\ \mathrm{IF}\,(\mathcal{B}[\![b]\!],\ f \circ \mathcal{D}[\![c]\!],\ id))
\end{aligned}$$

mit folgenden Hilfsfunktionen:

- $f \circ g$ ist die normale Funktionskomposition, nur dass sie $\bot$ propagiert:

$$(f \circ g)(\sigma) = \begin{cases} \bot & \text{falls } g(\sigma) = \bot \\ f(g(\sigma)) & \text{sonst} \end{cases}$$

- IF :: $((\Sigma \Rightarrow \mathbb{B}) \times (\Sigma \rightharpoonup \Sigma) \times (\Sigma \rightharpoonup \Sigma)) \Rightarrow (\Sigma \rightharpoonup \Sigma)$ wählt abhängig vom ersten Parameter einen der beiden anderen aus:

$$\mathrm{IF}\,(p,\ f,\ g)\,(\sigma) = \begin{cases} f(\sigma) & \text{falls } p(\sigma) = \mathbf{tt} \\ g(\sigma) & \text{falls } p(\sigma) = \mathbf{ff} \end{cases}$$

Zum FIX-Operator noch ein paar Überlegungen: Von der Big-Step-Semantik wissen wir, dass die Programme `while (`b`) do` c und `if (`b`) then` c`; while (`b`) do` c `else skip` äquivalent sind (Lem. 9). Demnach sollten auch die denotationalen Bedeutungen der beiden Programme gleich sein:

$$\begin{aligned}
\mathcal{D}[\![\texttt{while (}b\texttt{) do } c]\!] &= \mathcal{D}[\![\texttt{if (}b\texttt{) then } c\texttt{; while (}b\texttt{) do } c \texttt{ else skip}]\!] \\
&= \mathrm{IF}\,(\mathcal{B}[\![b]\!],\ \mathcal{D}[\![\texttt{while (}b\texttt{) do } c]\!] \circ \mathcal{D}[\![c]\!],\ id)
\end{aligned}$$

Dies entspricht der versuchten Definitionsgleichung für $\mathcal{D}[\![\texttt{while (}b\texttt{) do } c]\!]$ am Anfang dieses Kapitels. Sie drückt aus, dass $\mathcal{D}[\![\texttt{while (}b\texttt{) do } c]\!]$ ein Fixpunkt des Funktionals $\lambda f.\ \mathrm{IF}\,(\mathcal{B}[\![b]\!],\ f \circ \mathcal{D}[\![c]\!],\ id)$ sein soll. Wenn der Fixpunktoperator also wirklich einen Fixpunkt berechnet, dann erfüllt die denotationale Semantik-Definition diese Gleichung.

Die möglicherweise verschiedenen Fixpunkte des Funktionals unterscheiden sich nur auf Zuständen, für die das Programm nicht terminiert. Die fehlende Information über einen Endzustand in der Big-Step-Semantik äußert sich als Unterspezifikation in der denotationalen Semantik. Mit der Wahl des „undefiniertesten“ Fixpunkt drückt Undefiniertheit also Nichttermination aus. Dies entspricht der Nicht-Ableitbarkeit in der Big-Step-Semantik.

Beispiel 87. Für `while (x <= 0) do skip` ergibt sich folgendes Funktional F:

$$F(f) = \mathrm{IF}\,(\mathcal{B}[\![\texttt{x <= 0}]\!],\ f \circ \mathcal{D}[\![\texttt{skip}]\!],\ id) = \lambda\sigma.\ \begin{cases}\sigma & \text{falls } \sigma(\texttt{x}) > 0\\ f(\sigma) & \text{falls } \sigma(\texttt{x}) \leq 0\end{cases}$$

Die Fixpunktiteration von F ergibt:

$$\begin{aligned}
F^0(\bot) &= \bot\\
F^1(\bot)(\sigma) &= \begin{cases}\sigma & \text{falls } \sigma(\texttt{x}) > 0\\ \bot(\sigma) & \text{falls } \sigma(\texttt{x}) \leq 0\end{cases}\\
F^2(\bot)(\sigma) &= \begin{cases}\sigma & \text{falls } \sigma(\texttt{x}) > 0\\ F^1(\bot)(\sigma) & \text{falls } \sigma(\texttt{x}) \leq 0\end{cases} = \begin{cases}\sigma & \text{falls } \sigma(\texttt{x}) > 0\\ \bot(\sigma) & \text{falls } \sigma(\texttt{x}) \leq 0\end{cases} = F^1(\bot)(\sigma)\\
F^3(\bot)(\sigma) &= F(F^2(\bot))(\sigma) = F(F^1(\bot))(\sigma) = F^2(\bot)(\sigma) = F^1(\bot)(\sigma)
\end{aligned}$$

Mittels Induktion erhält man allgemein $F^{n+1}(\bot) = F^1(\bot)$ für alle n. Die Folge der Fixpunktiteration besteht nur aus den Elementen $F^0(\bot)$ und $F^1(\bot)$, der Grenzwert ist damit $F^1(\bot)$. Es gilt also:

$$\mathcal{D}[\![\texttt{while (x <= 0) do skip}]\!] = F^1(\bot) = \lambda\sigma.\ \begin{cases}\sigma & \text{falls } \sigma(\texttt{x}) > 0\\ \bot & \text{sonst}\end{cases}$$

7.2 Fixpunkttheorie

Im vorherigen Abschnitt haben wir den Fixpunktoperator FIX als Grenzwert der Fixpunktiteration postuliert. Es ist aber noch nicht klar, dass die Fixpunktiteration immer zu einem Grenzwert konvergiert und was diese Konvergenz sein soll. Ebenso ist bisher nur behauptet, dass die Fixpunktiteration wirklich den „undefiniertesten“ Fixpunkt liefert. Außerdem ist die Definition über die Fixpunktiteration noch immer sehr mit operationellen Details durchsetzt, die in der denotationalen Semantik nicht erwünscht sind.

Die Fixpunkttheorie liefert die mathematischen Begriffe und Techniken, um diese Probleme formal zu lösen: Man ordnet alle semantischen Objekte nach ihrem Informationsgehalt.

Definition 88 (Approximationsordnung). Seien f und g partielle Funktionen des Typs $\Sigma \rightharpoonup \Sigma$. f *approximiert* g ($f \sqsubseteq g$), falls für alle σ, σ' gilt:

$$\text{Wenn } f(\sigma) = \sigma', \text{ dann } g(\sigma) = \sigma'.$$

g muss also mindestens für all die Zustände definiert sein, für die f definiert ist, und in diesem Fall den gleichen Ergebniszustand haben. Ist f dagegen für einen Zustand σ nicht definiert, ist $g(\sigma)$ beliebig.

Lemma 89. $\sqsubseteq$ ist eine Halbordnung auf $\Sigma \rightharpoonup \Sigma$.

Beweis. Man muss Reflexivität, Antisymmetrie und Transitivität zeigen. Alle drei sind trivial. □

Beispiel 90. Seien

$$f_1(\sigma) = \sigma \qquad f_2(\sigma) = \begin{cases} \sigma & \text{falls } \sigma(\mathtt{x}) > 0 \\ \bot & \text{sonst} \end{cases} \qquad f_3(\sigma) = \begin{cases} \sigma & \text{falls } \sigma(\mathtt{x}) < 0 \\ \bot & \text{sonst} \end{cases} \qquad \text{und} \qquad f_4(\sigma) = \bot$$

Dann gilt $f_4 \sqsubseteq f_3 \sqsubseteq f_1$ und $f_4 \sqsubseteq f_2 \sqsubseteq f_1$, aber weder $f_2 \sqsubseteq f_3$, noch $f_3 \sqsubseteq f_2$. $\sqsubseteq$ ist also keine totale Ordnung.

Definition 91 (Kleinstes Element). Ein *kleinstes Element* einer geordneten Menge $(D, \sqsubseteq)$ ist ein Element $d \in D$, so dass für alle Elemente $d' \in D$ gilt: $d \sqsubseteq d'$. Bezogen auf unsere Informationsordnung $\sqsubseteq$ ist das kleinste Element das, das *keine Information* enthält; wir bezeichnen es mit $\bot$.

Kleinste Elemente sind, wenn sie existieren, eindeutig (wegen der Antisymmetrie).

Lemma 92. Die überall undefinierte Funktion $(\lambda\sigma.\ \bot)$ ist das kleinste Element von $(\Sigma \rightharpoonup \Sigma, \sqsubseteq)$.

Beweis. Sei f eine partielle Funktion $\Sigma \rightharpoonup \Sigma$. Zu zeigen: Wenn $(\lambda\sigma.\ \bot)(\sigma) = \sigma'$, dann $f(\sigma) = \sigma'$. $(\lambda\sigma.\ \bot)(\sigma)$ ist aber immer undefiniert, somit nie gleich σ'. Damit ist die Annahme nicht erfüllt und der Beweis trivial. □

Definition 93 (Obere Schranke). Ein Element $d \in D$ einer geordneten Menge $(D, \sqsubseteq)$ heißt *obere Schranke* einer Menge $Y \subseteq D$, falls für alle $d' \in Y$ gilt: $d' \sqsubseteq d$. d heißt *kleinste obere Schranke* von Y, falls d eine obere Schranke ist und für alle (anderen) oberen Schranken d' von Y gilt: $d \sqsubseteq d'$. Die kleinste obere Schranke von Y ist eindeutig, wenn sie existiert, und wird mit $\bigsqcup Y$ bezeichnet.

Beispiel 94. Seien

$$f_1(\sigma) = \begin{cases} \sigma & \text{falls } \sigma(\mathtt{x}) = 0 \\ \bot & \text{sonst} \end{cases} \qquad f_2(\sigma) = \begin{cases} \sigma & \text{falls } \sigma(\mathtt{y}) = 0 \\ \bot & \text{sonst} \end{cases} \qquad \text{und} \qquad f_3(\sigma) = \sigma$$

Dann gilt: $f_1 \sqsubseteq f_3$ und $f_2 \sqsubseteq f_3$, also ist f_3 obere Schranke von $Y = \{ f_1,\ f_2 \}$. Sei

$$f_4(\sigma) = \begin{cases} \sigma & \text{falls } \sigma(\mathtt{x}) = 0 \text{ oder } \sigma(\mathtt{y}) = 0 \\ \bot & \text{sonst} \end{cases}$$

Wegen $f_1 \sqsubseteq f_4$ und $f_2 \sqsubseteq f_4$ ist f_4 auch eine obere Schranke von Y. f_4 ist übrigens die kleinste obere Schranke:

Beweis. Sei g eine beliebige obere Schranke von Y. Dann gilt $f_1 \sqsubseteq g$ und $f_2 \sqsubseteq g$. Zu zeigen: $f_4 \sqsubseteq g$. Seien also σ, σ' beliebig mit $f_4(\sigma) = \sigma'$. Zu zeigen: $g(\sigma) = \sigma'$. Beweis durch Fallunterscheidung:

- Fall $\sigma(\mathtt{x}) = 0$: Dann $f_1(\sigma) = \sigma$ und $f_4(\sigma) = \sigma$, also $\sigma' = \sigma$. Wegen $f_1 \sqsubseteq g$ gilt auch $g(\sigma) = \sigma$.
- Fall $\sigma(\mathtt{y}) = 0$: Dann $f_2(\sigma) = \sigma$ und $f_4(\sigma) = \sigma$, also $\sigma' = \sigma$. Wegen $f_2 \sqsubseteq g$ gilt auch $g(\sigma) = \sigma$.
- Fall $\sigma(\mathtt{x}) \neq 0$ und $\sigma(\mathtt{y}) \neq 0$: Dann $f_4(\sigma) = \bot$, im Widerspruch zu $f_4(\sigma) = \sigma'$. □

Beispiel 95. Sei A eine beliebige Menge und $\mathfrak{P}(A)$ die Potenzmenge von A. $(\mathfrak{P}(A), \subseteq)$ ist eine Halbordnung, in der jede Menge $\mathfrak{B}$ von Teilmengen von A eine kleinste obere Schranken besitzt: Für $\mathfrak{B} \subseteq \mathfrak{P}(A)$ ist $\bigcup \mathfrak{B} = \{ a \in B \mid B \in \mathfrak{B} \}$ die kleinste obere Schranke.

Definition 96 (Kette). Eine Menge $Y \subseteq D$ heißt *Kette*, falls alle Elemente miteinander vergleichbar sind. Formal: Für alle $d, d' \in Y$ gilt: $d \sqsubseteq d'$ oder $d' \sqsubseteq d$.

Beispiel 97. Seien f_1, f_2, f_3 und f_4 wie im Beispiel 94. Dann ist $Y = \{ f_2,\ f_3,\ f_4 \}$ eine Kette mit $\bigsqcup Y = f_3$, weil $f_2 \sqsubseteq f_4 \sqsubseteq f_3$. $Z = \{ f_1,\ f_2,\ f_3,\ f_4 \}$ ist keine Kette, weil weder $f_1 \sqsubseteq f_2$ noch $f_2 \sqsubseteq f_1$.

Definition 98 (Kettenvollständige Halbordnung, ccpo). D heißt *kettenvollständige Halbordnung* (*chain-complete partial order*, *ccpo*), falls jede Kette in D eine kleinste obere Schranke in D besitzt.

Beispiel 99. Sei $<$ die normale Ordnung auf den rationalen Zahlen $\mathbb{Q}$ und Y die Menge aller geraden Zahlen. Dann ist Y eine Kette in $\mathbb{Q}$, da $0 < 2 < 4 < 6 < ...$, aber $\bigsqcup Y$ existiert nicht. Auch $Z = \{ x \in \mathbb{Q} \mid x < \sqrt{2} \}$ ist eine Kette in $\mathbb{Q}$, die keine kleinste obere Schranke in den rationalen Zahlen hat.

Lemma 100. $(\Sigma \rightharpoonup \Sigma,\ \sqsubseteq)$ ist eine kettenvollständige Halbordnung.

Beweis. Lem. 89 besagt, dass $(\Sigma \rightharpoonup \Sigma,\ \sqsubseteq)$ eine Halbordnung ist. Sei also Y eine Kette mit Elementen aus $\Sigma \rightharpoonup \Sigma$. Zu zeigen: Y hat eine kleinste obere Schranke. Definiere

$$\left(\bigsqcup Y\right)(\sigma) = \begin{cases} \sigma' & \text{falls es ein } f \in Y \text{ gibt mit } f(\sigma) = \sigma' \\ \bot & \text{sonst} \end{cases}$$

- $\bigsqcup Y$ ist wohldefiniert.:
  Seien f_1, f_2 aus Y mit $f_1(\sigma) = \sigma_1$ und $f_2(\sigma) = \sigma_2$. Da Y eine Kette ist, gilt $f_1 \sqsubseteq f_2$ oder $f_2 \sqsubseteq f_1$. Somit $\sigma_1 = \sigma_2$.
- $\bigsqcup Y$ ist eine obere Schranke von Y:
  Sei $f \in Y$. Zu zeigen: $f \sqsubseteq \bigsqcup Y$.
  Seien also σ, σ' beliebig mit $f(\sigma) = \sigma'$. Nach Definition von $\bigsqcup Y$ ist wie gefordert $(\bigsqcup Y)(\sigma) = \sigma'$.
- $\bigsqcup Y$ ist die kleinste obere Schranke von Y:
  Sei g obere Schranke von Y. Zu zeigen: $\bigsqcup Y \sqsubseteq g$.
  Seien σ, σ' beliebig mit $(\bigsqcup Y)(\sigma) = \sigma'$. Nach Definition von $\bigsqcup Y$ gibt es ein $f \in Y$ mit $f(\sigma) = \sigma'$.
  Da g obere Schranke von Y ist, gilt $f \sqsubseteq g$, also $g(\sigma) = \sigma'$. □

Definition 101 (Monotonie). Eine Funktion f zwischen zwei geordneten Mengen $(D, \sqsubseteq_D)$ und $(E, \sqsubseteq_E)$ heißt *monoton*, wenn für alle $d, d' \in D$ mit $d \sqsubseteq_D d'$ gilt: $f(d) \sqsubseteq_E f(d')$.

Monotone Funktionen respektieren also den Informationsgehalt: Je mehr Information man hinein gibt, desto mehr Information bekommt man heraus, und die neuen Information stehen nicht im Widerspruch zum vorherigen Ergebnis. Im Rahmen unserer Semantik besteht die geordnete Menge D selbst aus Funktionen; wir interessieren uns hier also für monotone Funktionale.

Beispiel 102. Das Funktional $\mathrm{IF}\,(p,\ f,\ g)$ ist monoton in g.
Seien $g_1 \sqsubseteq g_2$ beliebig. Zu zeigen: $\mathrm{IF}\,(p,\ f,\ g_1) \sqsubseteq \mathrm{IF}\,(p,\ f,\ g_2)$.
Seien also σ und σ' beliebig mit $\mathrm{IF}\,(p,\ f,\ g_1)\,(\sigma) = \sigma'$. Zu zeigen: $\mathrm{IF}\,(p,\ f,\ g_2)\,\sigma = \sigma'$.
Fallunterscheidung nach $p(\sigma)$:

- Fall $p(\sigma) = \mathbf{tt}$: Dann gilt $f(\sigma) = \sigma'$ und somit auch $\mathrm{IF}\,(p,\ f,\ g_2)\,\sigma = \sigma'$.
- Fall $p(\sigma) = \mathbf{ff}$: Dann gilt $g_1(\sigma) = \sigma'$. Wegen $g_1 \sqsubseteq g_2$ gilt auch $g_2(\sigma) = \sigma'$, also $\mathrm{IF}\,(p,\ f,\ g_2)\,\sigma = \sigma'$.

Analog erhält man, dass das Funktional $\mathrm{IF}\,(p,\ f,\ g)$ auch in f monoton ist.

Lemma 103. Die Komposition monotoner Funktionen ist monoton. Wenn f und g monoton sind, dann ist auch $f \circ g$ monoton.

Beweis. Seien $d, d' \in D$ beliebig mit $d \sqsubseteq d'$, dann gilt $g(d) \sqsubseteq g(d')$ wegen der Monotonie von g und mit der Monotonie von f auch $f(g(d)) \sqsubseteq f(g(d'))$. □

Erinnerung: Wie üblich bezeichnet $f(A)$ für $f :: X \Rightarrow Y$ und $A \subseteq X$ das Bild von A unter f, also $f(A) = \{f(x) \mid x \in A\}$.

Lemma 104 (Kettenerhalt unter monotonen Funktionen). Sei $f :: D \Rightarrow E$ eine monotone Funktion bezüglich der ccpos $(D, \sqsubseteq_D)$ und $(E, \sqsubseteq_E)$. Wenn Y eine Kette in $(D, \sqsubseteq_D)$ ist, dann ist $f(Y)$ eine Kette in E. Außerdem gilt:

$$\bigsqcup f(Y) \sqsubseteq_E f\left(\bigsqcup Y\right)$$

Beweis.

- $f(Y)$ ist eine Kette:
  Seien $e, e' \in f(Y)$ beliebig. Zu zeigen: $e \sqsubseteq_E e'$ oder $e' \sqsubseteq_E e$.
  Wegen $e, e' \in f(Y)$ gibt es d, d' aus Y mit $e = f(d)$ und $e' = f(d')$. Da Y eine Kette ist, gilt $d \sqsubseteq_D d'$ oder $d' \sqsubseteq_D d$. Da f monoton ist, folgt $f(d) \sqsubseteq_E f(d')$ oder $f(d') \sqsubseteq_E f(d)$.
- $\bigsqcup f(Y) \sqsubseteq_E f\,(\bigsqcup Y)$:
  Es genügt zu zeigen, dass $f\,(\bigsqcup Y)$ eine obere Schranke von $f(Y)$ ist, da $\bigsqcup f(Y)$ die kleinste obere Schranke von $f(Y)$ ist. Sei also $e \in f(Y)$ beliebig. Dann gibt es ein $d \in Y$ mit $f(d) = e$. Da Y eine Kette ist und $(D, \sqsubseteq_D)$ eine ccpo, ist $d \sqsubseteq_D \bigsqcup Y$. Da f monoton ist, folgt $f(d) \sqsubseteq_E f\,(\bigsqcup Y)$. Also ist $f(\bigsqcup Y)$ eine obere Schranke von $f(Y)$.

□

Beispiel 105. Sei $(\mathbb{N}^\infty, <)$ die geordnete Menge der natürlichen Zahlen zusammen mit Unendlich, und sei $(\mathbb{B}, <)$ die geordnete Menge der Wahrheitswerte ($\mathbf{ff} < \mathbf{tt}$). Sei $f :: \mathbb{N}^\infty \Rightarrow \mathbb{B}$ definiert durch $f(n) = (n = \infty)$. $(\mathbb{N}^\infty, <)$ und $(\mathbb{B}, <)$ sind ccpos und f ist monoton.

f erhält aber im Allgemeinen keine kleinsten oberen Schranken:

$$f\left(\bigsqcup \mathbb{N}\right) = f(\infty) = \mathbf{tt} \not\leq \mathbf{ff} = \bigsqcup \{\,\mathbf{ff}\,\} = \bigsqcup f(\mathbb{N})$$

Definition 106 (kettenstetig, strikt). Seien $(D, \sqsubseteq_D)$ und $(E, \sqsubseteq_E)$ ccpos und $f :: D \Rightarrow E$ eine monotone Funktion. Die Funktion f heißt *kettenstetig*, falls sie kleinste obere Schranken von Ketten erhält. Formal: Für alle nicht-leeren Ketten Y in $(D, \sqsubseteq_D)$ muss gelten:

$$\bigsqcup f(Y) = f\left(\bigsqcup Y\right)$$

Falls diese Gleichheit auch für leere Ketten gilt (d.h. $f(\bot) = \bot$), heißt f *strikt*.

Lemma 107. Die Komposition kettenstetiger Funktionen ist kettenstetig.

Beweis. Zu zeigen: Wenn f und g kettenstetig sind, dann ist es auch $f \circ g$.
Sei also Y eine nichtleere Kette. Dann ist auch $g(Y)$ eine nichtleere Kette nach Lem. 104. Damit gilt:

$$\bigsqcup (f \circ g)(Y) = \bigsqcup f(g(Y)) = f\left(\bigsqcup g(Y)\right) = f\left(g\left(\bigsqcup Y\right)\right) = (f \circ g)\left(\bigsqcup Y\right) \qquad \square$$

Theorem 108 (Knaster-Tarski-Fixpunktsatz). Sei $f :: D \Rightarrow D$ eine monotone, kettenstetige Funktion auf einer ccpo $(D, \sqsubseteq)$ mit kleinstem Element $\bot$. Dann hat f einen kleinsten Fixpunkt in D.

Insbesondere ist

$$\mathrm{FIX}\,(f) = \bigsqcup \{\, f^n(\bot) \mid n \in \mathbb{N} \,\}$$

dieser kleinste Fixpunkt.

Dieses Theorem gibt dem Grenzwert der Fixpunktiteration eine formale Grundlage: Er ist die kleinste obere Schranke der Fixpunktiterationsfolge.

Beweis. Für die Wohldefiniertheit ist zu zeigen, dass $\{\, f^n(\bot) \mid n \in \mathbb{N} \,\}$ eine Kette in D ist. Wegen der Transitivität von $\sqsubseteq$ genügt es, zu zeigen, dass $f^n(\bot) \sqsubseteq f^{n+1}(\bot)$ für alle n gilt. Beweis per Induktion über n:

- Fall $n = 0$: Da $\bot$ das kleinste Element von D ist, gilt: $f^0(\bot) = \bot \sqsubseteq f^1(\bot)$.
- Induktionsschritt: Zu zeigen: $f^{n+1}(\bot) \sqsubseteq f^{n+2}(\bot)$. Induktionsannahme: $f^n(\bot) \sqsubseteq f^{n+1}(\bot)$. Da f monoton ist, folgt aus der Induktionsannahme, dass $f(f^n(\bot)) \sqsubseteq f(f^{n+1}(\bot))$, was zu zeigen ist.

Die Fixpunkteigenschaft folgt aus folgender Gleichung:

$$\begin{aligned} f(\mathrm{FIX}\,(f)) &= f\left(\bigsqcup \{\, f^n(\bot) \mid n \in \mathbb{N} \,\}\right) = \bigsqcup \{\, f(f^n(\bot)) \mid n \in \mathbb{N} \,\} = \bigsqcup \{\, f^n(\bot) \mid n \geq 1 \,\} \\ &= \bigsqcup (\{\, f^n(\bot) \mid n \geq 1 \,\} \cup \{\, \bot \,\}) = \bigsqcup \{\, f^n(\bot) \mid n \in \mathbb{N} \,\} = \mathrm{FIX}\,(f) \end{aligned}$$

Jetzt fehlt noch der Beweis, dass $\mathrm{FIX}\,(f)$ der kleinste Fixpunkt von f ist.
Sei d ein Fixpunkt von f, also $f(d) = d$.
Wir müssen zeigen, dass $\mathrm{FIX}\,(f) = \bigsqcup \{\, f^n(\bot) \mid n \in \mathbb{N} \,\} \sqsubseteq d$ gilt. Da links die kleinste oberere Schranke steht, genügt es zu zeigen, dass für alle n gilt $f^n(\bot) \sqsubseteq d$, was wir per Induktion über n beweisen:

- Fall $n = 0$: Es gilt $f^0(\bot) = \bot \sqsubseteq d$, da $\bot$ das kleinste Element ist.
- Induktionsschritt: Zu zeigen: $f^{n+1}(\bot) \sqsubseteq d$. Induktionsannahme: $f^n(\bot) \sqsubseteq d$.
  Da f monoton ist, folgt aus der Induktionsannahme, dass $f(f^n(\bot)) \sqsubseteq f(d)$.
  Da d Fixpunkt ist, gilt $f^{n+1}(\bot) \sqsubseteq d$.

$\square$

7.3 Existenz des Fixpunkts für `while`

Mit Thm. 108 haben wir ein hinreichendes Kriterium dafür gefunden, dass Funktionen einen kleinsten Fixpunkt haben. Damit können wir das noch offene Problem der denotationalen Semantik für While, dass FIX für manche Funktionale nicht existieren könnte, lösen. Dazu müssen wir zeigen, dass nur monotone, kettenstetige Funktionale in der Definition von $\mathcal{D}\,[\![\_]\!]$ auftreten können.

Theorem 109. Seien $g, h :: \Sigma \rightharpoonup \Sigma$ beliebig. Das Funktional $F(f) = \mathrm{IF}\,(p,\ f \circ g,\ h)$ ist monoton und kettenstetig auf der ccpo $(\Sigma \rightharpoonup \Sigma, \sqsubseteq)$.

Beweis. Das Funktional F lässt sich in eine Hintereinanderausführung von Funktionalen umschreiben:

$$F = (\lambda f.\ \mathrm{IF}\,(p,\ f,\ h)) \circ (\lambda f.\ f \circ g)$$

Anmerkung: Die beiden Vorkommen $\circ$ bezeichnen verschiedene Funktionskompositionen! Das äußere ist die normale Hintereinanderausführung auf totalen Funktionalen, das innere die Komposition partieller Funktionen, die Undefiniertheit propagiert.

Nach Lem. 103 und 107 genügt es, zu zeigen, dass die beiden Teile monoton und kettenstetig sind.

- $\lambda f.\ f \circ g$ ist monoton: Sei $f_1 \sqsubseteq f_2$. Zu zeigen $f_1 \circ g \sqsubseteq f_2 \circ g$.

  Sei also $(f_1 \circ g)(\sigma) = \sigma'$. Zu zeigen: $(f_2 \circ g)(\sigma) = \sigma'$.

  Nach Definition gibt es ein σ'' mit $g(\sigma) = \sigma''$ und $f_1(\sigma'') = \sigma'$. Wegen $f_1 \sqsubseteq f_2$ gilt $f_2(\sigma'') = \sigma'$. Mit $g(\sigma) = \sigma''$ folgt $(f_2 \circ g)(\sigma) = \sigma'$ nach Definition.

- $\lambda f.\ f \circ g$ ist kettenstetig:
  Sei Y beliebige, nicht-leere Kette. Es genügt, folgende Ungleichung zu zeigen – die andere Richtung folgt schon aus Lem. 104.

  $$\left(\bigsqcup Y\right) \circ g \sqsubseteq \bigsqcup \{\, f \circ g \mid f \in Y \}$$

  Sei also $(\bigsqcup Y \circ g)(\sigma) = \sigma'$. Dann gibt es ein σ^* mit $g(\sigma) = \sigma^*$ und $(\bigsqcup Y)(\sigma^*) = \sigma'$. Nach Definition von $\bigsqcup Y$ gibt es ein $f \in Y$ mit $f(\sigma^*) = \sigma'$. Mit $g(\sigma) = \sigma^*$ gilt $(f \circ g)(\sigma) = \sigma'$. Wegen $f \circ g \in \{\, f \circ g \mid f \in Y \}$ ist $f \circ g \sqsubseteq \bigsqcup \{\, f \circ g \mid f \in Y \}$. Mit $(f \circ g)(\sigma) = \sigma'$ ist damit $(\bigsqcup \{\, f \circ g \mid f \in Y \})(\sigma) = \sigma'$ nach Definition.

- $\mathrm{IF}\,(p,\ f,\ g)$ ist monoton in f: Siehe Beispiel 102.

- $\mathrm{IF}\,(p,\ f,\ g)$ ist kettenstetig in f:
  Sei Y eine beliebige, nicht-leere Kette in $(\Sigma \rightharpoonup \Sigma, \sqsubseteq)$. Zu zeigen:

  $$\mathrm{IF}\left(p,\ \bigsqcup Y,\ g\right) \sqsubseteq \bigsqcup \{\, \mathrm{IF}\,(p,\ f,\ g) \mid f \in Y \,\}$$

  Sei also σ, σ' beliebig mit $\mathrm{IF}\,(p,\ \bigsqcup Y,\ g)\,(\sigma) = \sigma'$. Zu zeigen: $(\bigsqcup \{\, \mathrm{IF}\,(p,\ f,\ g) \mid f \in Y \,\})\,(\sigma) = \sigma'$.
  - Fall $p(\sigma) = \mathbf{tt}$: Damit $\mathrm{IF}\,(p,\ \bigsqcup Y,\ g)\,(\sigma) = \bigsqcup Y(\sigma) = \sigma'$. Nach Definition von $\bigsqcup Y$ gibt es also ein $f \in Y$ mit $f(\sigma) = \sigma'$. Somit gilt auch $\mathrm{IF}\,(p,\ f,\ g)\,(\sigma) = f(\sigma)$, da $p(\sigma) = \mathbf{tt}$. Da $\mathrm{IF}\,(p,\ f,\ g) \in \{\, \mathrm{IF}\,(p,\ f,\ g) \mid f \in Y \,\}$, ist $\mathrm{IF}\,(p,\ f,\ g) \sqsubseteq \bigsqcup \{\, \mathrm{IF}\,(p,\ f,\ g) \mid f \in Y \,\}$. Mit $f(\sigma) = \sigma'$ folgt die Behauptung.
  - Fall $p(\sigma) = \mathbf{ff}$: Damit $\mathrm{IF}\,(p,\ \bigsqcup Y,\ g)\,(\sigma) = g(\sigma) = \sigma'$. Da Y nicht leer ist, gibt es ein $f \in Y$. Dann ist $\mathrm{IF}\,(p,\ f,\ g)\,(\sigma) = g(\sigma) = \sigma'$ nach Definition. Da $\mathrm{IF}\,(p,\ f,\ g) \in \{\, \mathrm{IF}\,(p,\ f,\ g) \mid f \in Y \,\}$, ist $\mathrm{IF}\,(p,\ f,\ g) \sqsubseteq \bigsqcup \{\, \mathrm{IF}\,(p,\ f,\ g) \mid f \in Y \,\}$. Mit $g(\sigma) = \sigma'$ folgt die Behauptung. □

Damit ist $\mathcal{D}\,[\![\_]\!]$ durch Def. 86 wohldefiniert: In der kritischen Definitionsgleichung für `while (`b`) do` c

$$\mathcal{D}\,[\![\texttt{while (}b\texttt{) do }c]\!] = \mathrm{FIX}\,(\lambda f.\ \mathrm{IF}\,(\mathcal{B}\,[\![b]\!],\ f \circ \mathcal{D}\,[\![c]\!],\ id))$$

existiert der Fixpunkt immer nach Thm. 108, denn das Funktional $F(f) = \mathrm{IF}\,(\mathcal{B}\,[\![b]\!],\ f \circ \mathcal{D}\,[\![c]\!],\ id)$ ist nach Thm. 109 monoton und kettenstetig.

7.4 Bezug zur operationalen Semantik

Für While haben wir nun zwei operationale Semantiken und eine denotationale. Von den operationalen ist bekannt, dass sie das gleiche terminierende Verhalten definieren (Kor. 23), für unendliche Ausführungen haben wir das in der Übung untersucht. Nun stellt sich natürlich die Frage, in welchem Verhältnis die denotationale Semantik zu diesen beiden steht.

Lemma 110. Wenn $\langle c,\ \sigma\rangle \Downarrow \sigma'$, dann $\mathcal{D}[\![c]\!]\,\sigma = \sigma'$.

Beweis. Induktion über $\langle c,\ \sigma\rangle \Downarrow \sigma'$ (vgl. Def. 7)

- Fall SKIP_{BS}: Zu zeigen: $\mathcal{D}[\![\texttt{skip}]\!]\,\sigma = \sigma$. Nach Definition.
- Fall ASS_{BS}: Zu zeigen: $\mathcal{D}[\![x\ \texttt{:=}\ a]\!]\,\sigma = \sigma[x \mapsto \mathcal{A}[\![a]\!]\,\sigma]$. Nach Definition.
- Fall SEQ_{BS}: Induktionsannahmen: $\mathcal{D}[\![c_1]\!]\,\sigma = \sigma'$ und $\mathcal{D}[\![c_2]\!]\,\sigma' = \sigma''$.
  Zu zeigen: $\mathcal{D}[\![c_1\,\texttt{;}\ c_2]\!]\,\sigma = \sigma''$

  Nach Definition gilt $\mathcal{D}[\![c_1\,\texttt{;}\ c_2]\!]\,\sigma = (\mathcal{D}[\![c_2]\!] \circ \mathcal{D}[\![c_1]\!])(\sigma)$.
  Mit den Induktionsannahmen folgt $(\mathcal{D}[\![c_2]\!] \circ \mathcal{D}[\![c_1]\!])(\sigma) = \sigma''$.
- Fall IFTT_{BS}: Induktionsannahmen: $\mathcal{B}[\![b]\!]\,\sigma = \mathbf{tt}$ und $\mathcal{D}[\![c_1]\!]\,\sigma = \sigma'$.
  Zu zeigen: $\mathcal{D}[\![\texttt{if (}b\texttt{) then}\ c_1\ \texttt{else}\ c_2]\!]\,\sigma = \sigma'$.

$$\mathcal{D}[\![\texttt{if (}b\texttt{) then}\ c_1\ \texttt{else}\ c_2]\!]\,\sigma = \text{IF}\left(\mathcal{B}[\![b]\!],\ \mathcal{D}[\![c_1]\!],\ \mathcal{D}[\![c_2]\!]\right)(\sigma) = \mathcal{D}[\![c_1]\!]\,\sigma = \sigma'$$

- Fall IFFF_{BS}: Analog zu IFTT_{BS}.
- Fall $\text{WHILETT}_{\text{BS}}$:
  Induktionsannahmen: $\mathcal{B}[\![b]\!]\,\sigma = \mathbf{tt}$, $\mathcal{D}[\![c]\!]\,\sigma = \sigma'$ und $\mathcal{D}[\![\texttt{while (}b\texttt{) do}\ c]\!]\,\sigma' = \sigma''$.
  Zu zeigen: $\mathcal{D}[\![\texttt{while (}b\texttt{) do}\ c]\!]\,\sigma = \sigma''$.

$$\begin{aligned}
\mathcal{D}[\![\texttt{while (}b\texttt{) do}\ c]\!]\,\sigma &= \text{FIX}\left(\lambda f.\ \text{IF}\left(\mathcal{B}[\![b]\!],\ f \circ \mathcal{D}[\![c]\!],\ id\right)\right)(\sigma) \\
\text{(Fixpunkteigenschaft)} &= \text{IF}\left(\mathcal{B}[\![b]\!],\ \text{FIX}\left(\lambda f.\ \text{IF}\left(\mathcal{B}[\![b]\!],\ f \circ \mathcal{D}[\![c]\!],\ id\right)\right) \circ \mathcal{D}[\![c]\!],\ id\right)(\sigma) \\
&= \left(\text{FIX}\left(\lambda f.\ \text{IF}\left(\mathcal{B}[\![b]\!],\ f \circ \mathcal{D}[\![c]\!],\ id\right)\right) \circ \mathcal{D}[\![c]\!]\right)(\sigma) \\
&= \left(\mathcal{D}[\![\texttt{while (}b\texttt{) do}\ c]\!] \circ \mathcal{D}[\![c]\!]\right)(\sigma) = \sigma''
\end{aligned}$$

- Fall $\text{WHILEFF}_{\text{BS}}$: Induktionsannahme: $\mathcal{B}[\![b]\!]\,\sigma = \mathbf{ff}$. Zu zeigen: $\mathcal{D}[\![\texttt{while (}b\texttt{) do}\ c]\!]\,\sigma = \sigma$.

$$\begin{aligned}
\mathcal{D}[\![\texttt{while (}b\texttt{) do}\ c]\!]\,\sigma &= \text{FIX}\left(\lambda f.\ \text{IF}\left(\mathcal{B}[\![b]\!],\ f \circ \mathcal{D}[\![c]\!],\ id\right)\right)(\sigma) \\
&= \text{IF}\left(\mathcal{B}[\![b]\!],\ \text{FIX}\left(\lambda f.\ \text{IF}\left(\mathcal{B}[\![b]\!],\ f \circ \mathcal{D}[\![c]\!],\ id\right)\right) \circ \mathcal{D}[\![c]\!],\ id\right)(\sigma) = id(\sigma) = \sigma \square
\end{aligned}$$

Lemma 111. Wenn $\mathcal{D}[\![c]\!]\,\sigma = \sigma'$, dann $\langle c,\ \sigma\rangle \Downarrow \sigma'$.

Beweis. Induktion über c (σ, σ' beliebig):

- Fall $c = \texttt{skip}$: Zu zeigen: Wenn $\mathcal{D}[\![\texttt{skip}]\!]\,\sigma = \sigma'$, dann $\langle \texttt{skip},\ \sigma\rangle \Downarrow \sigma'$.

  Aus der Voraussetzung folgt $\sigma' = id(\sigma) = \sigma$, damit die Behauptung nach Regel SKIP_{BS}.
- Fall $c = x\ \texttt{:=}\ a$: Zu zeigen: Wenn $\mathcal{D}[\![x\ \texttt{:=}\ a]\!]\,\sigma = \sigma'$, dann $\langle x\ \texttt{:=}\ a,\ \sigma\rangle \Downarrow \sigma'$. Aus der Voraussetzung folgt $\sigma' = \sigma[x \mapsto \mathcal{A}[\![a]\!]\,\sigma]$. ASS_{BS} liefert dann die Behauptung.
- Fall $c = c_1\,\texttt{;}\ c_2$:
  Induktionsannahmen (für beliebige σ, σ'):
  Wenn $\mathcal{D}[\![c_1]\!]\,\sigma = \sigma'$, dann $\langle c_1,\ \sigma\rangle \Downarrow \sigma'$. Wenn $\mathcal{D}[\![c_2]\!]\,\sigma = \sigma'$, dann $\langle c_2,\ \sigma\rangle \Downarrow \sigma'$.

  Zu zeigen: Wenn $\mathcal{D}[\![c_1\,\texttt{;}\ c_2]\!]\,\sigma = \sigma'$, dann $\langle c_1\,\texttt{;}\ c_2,\ \sigma\rangle \Downarrow \sigma'$.

  Wegen $\mathcal{D}[\![c_1\,\texttt{;}\ c_2]\!]\,\sigma = (\mathcal{D}[\![c_2]\!] \circ \mathcal{D}[\![c_1]\!])(\sigma) = \sigma'$ gibt es ein σ^* mit $\mathcal{D}[\![c_1]\!]\,\sigma = \sigma^*$ und $\mathcal{D}[\![c_2]\!]\,\sigma^* = \sigma'$.
  Mit den Induktionsannahmen gilt also $\langle c_1,\ \sigma\rangle \Downarrow \sigma^*$ und $\langle c_2,\ \sigma^*\rangle \Downarrow \sigma'$. Damit folgt die Behauptung mit Regel SEQ_{BS}.

- Fall $c = \texttt{if (}b\texttt{) then } c_1 \texttt{ else } c_2$:
  Induktionsannahmen: Wenn $\mathcal{D}[\![c_1]\!]\sigma = \sigma'$, dann $\langle c_1,\ \sigma\rangle \Downarrow \sigma'$. Wenn $\mathcal{D}[\![c_2]\!]\sigma = \sigma'$, dann $\langle c_2,\ \sigma\rangle \Downarrow \sigma'$.
  Zu zeigen: Wenn $\mathcal{D}[\![\texttt{if (}b\texttt{) then } c_1 \texttt{ else } c_2]\!]\sigma = \sigma'$, dann $\langle \texttt{if (}b\texttt{) then } c_1 \texttt{ else } c_2,\ \sigma\rangle \Downarrow \sigma'$.

  Fallunterscheidung nach $\mathcal{B}[\![b]\!]\sigma$:
  - Fall $\mathcal{B}[\![b]\!]\sigma = \mathbf{tt}$: Dann gilt:

    $$\mathcal{D}[\![\texttt{if (}b\texttt{) then } c_1 \texttt{ else } c_2]\!]\sigma = \mathrm{IF}\left(\mathcal{B}[\![b]\!],\ \mathcal{D}[\![c_1]\!],\ \mathcal{D}[\![c_2]\!]\right)(\sigma) = \mathcal{D}[\![c_1]\!]\sigma = \sigma'$$

    Mit der Induktionsannahme folgt $\langle c_1,\ \sigma\rangle \Downarrow \sigma'$ und daraus zusammen mit $\mathcal{B}[\![b]\!]\sigma = \mathbf{tt}$ die Behauptung nach Regel $\textsc{IfTT}_{\mathrm{BS}}$.
  - Fall $\mathcal{B}[\![b]\!]\sigma = \mathbf{ff}$: Analog.

- Fall $c = \texttt{while (}b\texttt{) do } c$:
  Induktionsannahme I (für beliebige σ, σ'): Wenn $\mathcal{D}[\![c]\!]\sigma = \sigma'$, dann $\langle c,\ \sigma\rangle \Downarrow \sigma'$.
  Zu zeigen: Wenn $\mathcal{D}[\![\texttt{while (}b\texttt{) do } c]\!]\sigma = \sigma'$, dann $\langle \texttt{while (}b\texttt{) do } c,\ \sigma\rangle \Downarrow \sigma'$.

  Sei $F(f) = \mathrm{IF}\left(\mathcal{B}[\![b]\!],\ f \circ \mathcal{D}[\![c]\!],\ id\right)$ das Funktional für die Schleife. Dann gilt nach Thm. 108:

  $$\mathcal{D}[\![\texttt{while (}b\texttt{) do } c]\!]\sigma = \mathrm{FIX}\,(F)\,(\sigma) = \left(\bigsqcup \{\, F^n(\bot) \mid n \in \mathbb{N}\,\}\right)(\sigma) = \sigma'$$

  Nach Definition von $\bigsqcup \{\, F^n(\bot) \mid n \in \mathbb{N}\,\}$ gibt es ein $n \in \mathbb{N}$ mit $F^n(\bot)(\sigma) = \sigma'$.

  Behauptung: Wenn $F^n(\bot)(\sigma) = \sigma'$, dann $\langle \texttt{while (}b\texttt{) do } c,\ \sigma\rangle \Downarrow \sigma'$.
  Beweis duch Induktion über n (σ, σ' beliebig):
  - Basisfall $n = 0$: Zu zeigen: Wenn $F^0(\bot)(\sigma) = \sigma'$, dann $\langle \texttt{while (}b\texttt{) do } c,\ \sigma\rangle \Downarrow \sigma'$.
    Trivial, da Voraussetzung nicht erfüllt: $F^0(\bot)(\sigma) = \bot(\sigma) = \bot$.
  - Induktionsschritt $n + 1$:
    Induktionsannahme II (σ, σ' beliebig): Wenn $F^n(\bot)(\sigma) = \sigma'$, dann $\langle \texttt{while (}b\texttt{) do } c,\ \sigma\rangle \Downarrow \sigma'$.
    Zu zeigen: Wenn $F^{n+1}(\bot)(\sigma) = \sigma'$, dann $\langle \texttt{while (}b\texttt{) do } c,\ \sigma\rangle \Downarrow \sigma'$.
    Beweis von $\langle \texttt{while (}b\texttt{) do } c,\ \sigma\rangle \Downarrow \sigma'$ durch Fallunterscheidung nach $\mathcal{B}[\![b]\!]\sigma$:
    - Fall $\mathcal{B}[\![b]\!]\sigma = \mathbf{tt}$: Dann gilt:

      $$\sigma' = F^{n+1}(\bot)(\sigma) = F(F^n(\bot))(\sigma) = \mathrm{IF}\left(\mathcal{B}[\![b]\!],\ F^n(\bot) \circ \mathcal{D}[\![c]\!],\ id\right)(\sigma) = (F^n(\bot) \circ \mathcal{D}[\![c]\!])(\sigma)$$

      Damit gibt es ein σ^* mit $\mathcal{D}[\![c]\!]\sigma = \sigma^*$ und $F^n(\bot)(\sigma^*) = \sigma'$. Aus $\mathcal{D}[\![c]\!]\sigma = \sigma^*$ folgt nach Induktionsannahme I, dass $\langle c,\ \sigma\rangle \Downarrow \sigma^*$. Aus $F^n(\bot)(\sigma^*) = \sigma'$ folgt nach Induktionsannahme II, dass $\langle \texttt{while (}b\texttt{) do } c,\ \sigma^*\rangle \Downarrow \sigma'$. Zusammen folgt die Behauptung nach Regel $\textsc{WhileTT}_{\mathrm{BS}}$.
    - Fall $\mathcal{B}[\![b]\!]\sigma = \mathbf{ff}$: Dann gilt:

      $$F^{n+1}(\bot)(\sigma) = F(F^n(\bot))(\sigma) = \mathrm{IF}\left(\mathcal{B}[\![b]\!],\ F^n(\bot) \circ \mathcal{D}[\![c]\!],\ id\right)(\sigma) = id(\sigma) = \sigma$$

      Also $\sigma' = \sigma$. Mit $\mathcal{B}[\![b]\!]\sigma = \mathbf{ff}$ folgt die Behauptung nach Regel $\textsc{WhileFF}_{\mathrm{BS}}$. □

Theorem 112 (Adäquatheit, Äquivalenz von operationaler und denotationaler Semantik). Für alle c, σ und σ' gilt $\langle c,\ \sigma\rangle \Downarrow \sigma'$ gdw. $\langle c,\ \sigma\rangle \xrightarrow{*}_1 \langle \texttt{skip},\ \sigma'\rangle$ gdw. $\mathcal{D}[\![c]\!]\sigma = \sigma'$.

Was hat man nun von der denotationalen Semantik? Viele Aussagen über die operationale Semantik sind (dank des Adäquatheitstheorems) in der denotationalen Semantik viel einfacher zu beweisen. Man braucht für vieles nicht mehr Induktionen über Ableitungsbäume zu führen! Beispielsweise werden die Determinismusbeweise (Thm. 10 und Kor. 18) hinfällig, da dies direkt aus dem Funktionscharakter von $\mathcal{D}[\![\_]\!]$ folgt. Auch das Schleifenabwicklungslemma 9 wird trivial.

Aus Kompositionalität und Adäquatheit kann man noch mehr Nutzen ziehen: Zwei Programme c_1 und c_2 mit der gleichen denotationalen Semantik ($\mathcal{D}[\![c_1]\!] = \mathcal{D}[\![c_2]\!]$) können jederzeit gegeneinander ausgetauscht werden – sogar beliebig innerhalb anderer Sprachkonstrukte.

Definition 113 (Kontext). Ein Kontext ist ein While-Programm mit einem Loch []. Formal ist ein Kontext (Variablenkonvention K) durch folgende Grammatik gegeben:

$$\begin{array}{lrcl} \mathsf{Context} & K & ::= & [] \mid K\texttt{;}\ c \mid c\texttt{;}\ K \mid \texttt{if (}b\texttt{) then }K\texttt{ else }c \mid \\ & & & \texttt{if (}b\texttt{) then }c\texttt{ else }K \mid \texttt{while (}b\texttt{) do }K \end{array}$$

Die Kontextfüll-Funktion _ [_] ersetzt das Loch [] im Kontext K durch das übergebene Programm c':

$$\begin{array}{rcl} []\,[c'] & = & c' \\ (K\texttt{;}\ c)\,[c'] & = & K\,[c']\texttt{;}\ c \\ (c\texttt{;}\ K)\,[c'] & = & c\texttt{;}\ \texttt{(}K\,[c']\texttt{)} \\ (\texttt{if (}b\texttt{) then }K\texttt{ else }c)\,[c'] & = & \texttt{if (}b\texttt{) then }K\,[c']\texttt{ else }c \\ (\texttt{if (}b\texttt{) then }c\texttt{ else }K)\,[c'] & = & \texttt{if (}b\texttt{) then }c\texttt{ else (}K\,[c']\texttt{)} \\ (\texttt{while (}b\texttt{) do }K)\,[c'] & = & \texttt{while (}b\texttt{) do (}K\,[c']\texttt{)} \end{array}$$

Definition 114 (Semantik eines Kontexts). Die Semantik $\mathcal{K}[\![K]\!]$ eines Kontexts K ist vom Typ $(\Sigma \rightharpoonup \Sigma) \Rightarrow (\Sigma \rightharpoonup \Sigma)$ und rekursiv über K definiert:

$$\begin{array}{rcl} \mathcal{K}[\![[]]\!]\,f & = & f \\ \mathcal{K}[\![K\texttt{;}\ c]\!]\,f & = & \mathcal{D}[\![c]\!] \circ \mathcal{K}[\![K]\!]\,f \\ \mathcal{K}[\![c\texttt{;}\ K]\!]\,f & = & \mathcal{K}[\![K]\!]\,f \circ \mathcal{D}[\![c]\!] \\ \mathcal{K}[\![\texttt{if (}b\texttt{) then }K\texttt{ else }c]\!]\,f & = & \mathrm{IF}\,(\mathcal{B}[\![b]\!],\ \mathcal{K}[\![K]\!]\,f,\ \mathcal{D}[\![c]\!]) \\ \mathcal{K}[\![\texttt{if (}b\texttt{) then }c\texttt{ else }K]\!]\,f & = & \mathrm{IF}\,(\mathcal{B}[\![b]\!],\ \mathcal{D}[\![c]\!],\ \mathcal{K}[\![K]\!]\,f) \\ \mathcal{K}[\![\texttt{while (}b\texttt{) do }K]\!]\,f & = & \mathrm{FIX}\,(\lambda g.\ \mathrm{IF}\,(\mathcal{B}[\![b]\!],\ g \circ \mathcal{K}[\![K]\!]\,f,\ id)) \end{array}$$

Theorem 115 (Kompositionalität).
Für alle Kontexte K und Programme c gilt $\mathcal{D}[\![K\,[c]]\!] = \mathcal{K}[\![K]\!]\,(\mathcal{D}[\![c]\!])$

Beweis. Induktion über K und ausrechnen. Beispielhaft der Fall für `while (`b`) do `K:
Induktionsannahmen: $\mathcal{D}[\![K\,[c]]\!] = \mathcal{K}[\![K]\!]\,(\mathcal{D}[\![c]\!])$ für alle c.
Zu zeigen: $\mathcal{D}[\![(\texttt{while (}b\texttt{) do }K)\,[c]]\!] = \mathcal{K}[\![\texttt{while (}b\texttt{) do }K]\!]\,(\mathcal{D}[\![c]\!])$

$$\begin{array}{rcl} \mathcal{D}[\![(\texttt{while (}b\texttt{) do }K)\,[c]]\!] = \mathcal{D}[\![\texttt{while (}b\texttt{) do (}K\,[c]\,\texttt{)}]\!] & = & \mathrm{FIX}\,(\lambda f.\ \mathrm{IF}\,(\mathcal{B}[\![b]\!],\ f \circ \mathcal{D}[\![K\,[c]]\!],\ id)) \\ & = & \mathrm{FIX}\,(\lambda f.\ \mathrm{IF}\,(\mathcal{B}[\![b]\!],\ f \circ \mathcal{K}[\![K]\!]\,(\mathcal{D}[\![c]\!]),\ id)) \\ & = & \mathcal{K}[\![\texttt{while (}b\texttt{) do }K]\!]\,(\mathcal{D}[\![c]\!]) \end{array}$$

□

Korollar 116. Zwei Programme c_1 und c_2 mit gleicher Semantik sind in allen Kontexten austauschbar: Wenn $\mathcal{D}[\![c_1]\!] = \mathcal{D}[\![c_2]\!]$, dann $\mathcal{D}[\![K\,[c_1]]\!] = \mathcal{D}[\![K\,[c_2]]\!]$.

Beweis. Nach Thm. 115 gilt $\mathcal{D}[\![K\,[c_1]]\!] = \mathcal{K}[\![K]\!]\,(\mathcal{D}[\![c_1]\!]) = \mathcal{K}[\![K]\!]\,(\mathcal{D}[\![c_2]\!]) = \mathcal{D}[\![K\,[c_2]]\!]$. □

Beispiel 117. $c_1 =$ `while (`b`) do `c und $c_2 =$ `if (`b`) then `c`; while (`b`) do `c` else skip` haben die gleiche denotationale Semantik. Damit kann ein Compiler in allen Programmen c_2 durch c_1 ersetzen (oder umgekehrt), ohne das Verhalten zu ändern.

Übung: Beweisen Sie ohne Verwendung der denotationalen Semantik direkt, dass man jederzeit Schleifen abwickeln darf.

7.5 Continuation-style denotationale Semantik

In Kapitel 6.4 haben wir bereits die Erweiterung While_X von While um Ausnahmen und deren Behandlung mit den Anweisungen `raise` X und `try` c_1 `catch` X c_2 betrachtet. Bei der Big-Step-Semantik haben wir ein zusätzliches Rückgabeflag eingeführt, um normale und außergewöhnliche Termination zu unterscheiden. Entsprechend mussten auch alle bestehenden Regeln sorgfältig angepasst und die Möglichkeit für eine Ausnahme in jedem Schritt eigens behandelt werden. In der Small-Step-Semantik musste dazu eine eigene `raise` _-Regel für alle zusammengesetzten Konstrukte (bei uns nur c_1 `;` c_2) eingeführt werden.

Ganz analog zur Big-Step-Semantik ließe sich auch die denotationale Semantik für While um Exceptions erweitern. Allerdings ist dieser Formalismus insgesamt nicht zufrieden stellend, da Ausnahmen nun einmal die Ausnahme sein und deswegen nicht explizit durch jedes Programmkonstrukt durchgeschleift werden sollten. Dafür gibt es Fortsetzungen (continuations), die die Semantik (d.h. den Effekt) der Ausführung des restlichen Programms beschreiben. Eine einfache Form der Fortsetzungen haben wir schon in der Übung zur Goto-Erweiterung bei der Small-Step-Semantik gesehen, bei der aber die Fortsetzung noch syntaktisch als Anweisungsliste repräsentiert war.

Definition 118 (Fortsetzung, continuation). Eine *Fortsetzung* (*continuation*) ist eine partielle Funktion auf Zuständen, die das Ergebnis der Ausführung des restlichen Programms von einem Zustand aus beschreibt. $\mathsf{Cont} = \Sigma \rightharpoonup \Sigma$ bezeichne die Menge aller Fortsetzungen.[5]

Statt nun anhand eines Flags einem umgebenden Programmkonstrukts die Auswahl der restlichen Berechnung zu überlassen, soll jedes Konstrukt direkt auswählen, ob es normal oder mit einer (und welcher) Ausnahmebehandlung weitergehen soll. Dazu muss die Semantik der restlichen normalen bzw. außergewöhnlichen Auswertung direkt bei der Definition eines Konstrukts als Fortsetzung zur Verfügung stehen. Zum Beispiel wählt `raise` X die Fortsetzung „Ausnahmebehandlung für X“ und `skip` die Fortsetzung „normale Ausführung“ aus. Alle möglichen Fortsetzungen müssen also als Parameter an die Semantikfunktion $\mathcal{C}[\![\_]\!]$ gegeben werden, die damit den Typ

$$\mathsf{Com} \Rightarrow \underbrace{\mathsf{Cont}}_{\text{normale Fortsetzung}} \Rightarrow (\underbrace{\mathsf{Xcp} \Rightarrow \mathsf{Cont}}_{\text{Ausnahmebehandlung}}) \Rightarrow \mathsf{Cont}$$

hat. Intuitiv bedeutet $\mathcal{C}[\![c]\!]\, s\ S\ \sigma$ also: Führe c im Zustand σ aus und setze mit s normal bzw. mit $S(X)$ bei der Ausnahme X fort. S ist dabei die “Umgebung” der Ausnahmebehandlungen, die von `try _ catch _ _` blockartig geändert wird. Formal:

$$\begin{aligned}
\mathcal{C}[\![\texttt{skip}]\!]\, s\ S &= s \\
\mathcal{C}[\![x\ \texttt{:=}\ a]\!]\, s\ S &= \lambda\sigma.\ s(\sigma[x \mapsto \mathcal{A}[\![a]\!]\,\sigma]) \\
\mathcal{C}[\![c_1\texttt{;}\ c_2]\!]\, s\ S &= \mathcal{C}[\![c_1]\!]\,(\mathcal{C}[\![c_2]\!]\, s\ S)\ S \\
\mathcal{C}[\![\texttt{if (}b\texttt{) then } c_1 \texttt{ else } c_2]\!]\, s\ S &= \mathrm{IF}\,(\mathcal{B}[\![b]\!],\ \mathcal{C}[\![c_1]\!]\, s\ S,\ \mathcal{C}[\![c_2]\!]\, s\ S) \\
\mathcal{C}[\![\texttt{while (}b\texttt{) do } c]\!]\, s\ S &= \mathrm{FIX}\,(\lambda f.\ \mathrm{IF}\,(\mathcal{B}[\![b]\!],\ \mathcal{C}[\![c]\!]\, f\ S,\ s)) \\
\mathcal{C}[\![\texttt{raise } X]\!]\, s\ S &= S(X) \\
\mathcal{C}[\![\texttt{try } c_1 \texttt{ catch } X\ c_2]\!]\, s\ S &= \mathcal{C}[\![c_1]\!]\, s\ (S[X \mapsto \mathcal{C}[\![c_2]\!]\, s\ S])
\end{aligned}$$

Für ein Programm verwendet man üblicherweise die anfänglichen Fortsetzungen $s_0 = id$ und $S_0 = \bot$, sofern man Nichttermination und unbehandelte Ausnahmen nicht unterscheiden möchte. Ansonsten muss man Cont auf eine allgemeinere Antwortmenge wie z.B. $\Sigma \rightharpoonup (\mathsf{Xcp}' \times \Sigma)$ bei der Big-Step-Semantik verallgemeinern – in diesem Fall wären dann $s_0(\sigma) = (\texttt{None}, \sigma)$ und $S_0(X)(\sigma) = (X, \sigma)$.

[5] Genau genommen ist es nicht wesentlich, was der Rückgabetyp einer Fortsetzung selbst ist. In unserem Fall werden das meist Zustände (Σ) sein, im Allgemeinen nimmt man aber oft den abstrakten Typ *Answer*.

Beispiel 119. Sei $c = \texttt{try (x := 2; raise Xcpt; x := 3) catch Xcpt x := 4}$.

$$\begin{aligned}
\mathcal{C}\,[\![c]\!]\,s\;S\;\sigma &= \mathcal{C}\,[\![\texttt{x := 2; raise Xcpt; x := 3}]\!]\,s\,(\overbrace{S[\texttt{Xcpt} \mapsto \mathcal{C}\,[\![\texttt{x := 4}]\!]\,s\;S]}^{=S'})\;\sigma \\
&= \mathcal{C}\,[\![\texttt{x := 2}]\!]\,(\mathcal{C}\,[\![\texttt{raise Xcpt; x := 3}]\!]\,s\;S')\;S'\;\sigma \\
&= \mathcal{C}\,[\![\texttt{raise Xcpt; x := 3}]\!]\,s\;S'\;(\sigma[\texttt{x} \mapsto 2]) \\
&= \mathcal{C}\,[\![\texttt{raise Xcpt}]\!]\,(\mathcal{C}\,[\![\texttt{x := 3}]\!]\,s\;S')\;S'\;(\sigma[\texttt{x} \mapsto 2]) \\
&= S'(\texttt{Xcpt})\;(\sigma[\texttt{x} \mapsto 2]) = \mathcal{C}\,[\![\texttt{x := 4}]\!]\,s\;S\;(\sigma[\texttt{x} \mapsto 2]) = s(\sigma[\texttt{x} \mapsto 4])
\end{aligned}$$

Damit gilt für $s = s_0 = id$ und $S = S_0 = \bot$: $\mathcal{C}\,[\![c]\!]\,s\;S\;\sigma = \mathcal{C}\,[\![c]\!]\,id\;\bot\;\sigma = \sigma[\texttt{x} \mapsto 4]$.

Noch ein paar Anmerkungen zur Continuation-Semantik $\mathcal{C}\,[\![\_]\!]$:

- $\mathcal{C}\,[\![\texttt{skip}]\!]$ ist nicht mehr einfach die Identität, sondern die Fortsetzung. Das tritt analog auch bei $\texttt{while (}b\texttt{) do }c$ auf.
- Die Reihenfolge von c_1 und c_2 in $\mathcal{C}\,[\![c_1\texttt{;}\ c_2]\!]$ ist nicht mehr wie bei $\mathcal{D}\,[\![c_1\texttt{;}\ c_2]\!]$ vertauscht.
- Das Funktional für den Fixpunktoperator in der Gleichung $\texttt{while (}b\texttt{) do }c$ entstammt der Rekursionsgleichung

  $$\mathcal{C}\,[\![\texttt{while (}b\texttt{) do }c]\!]\,s\;S = \mathrm{IF}\,(\mathcal{B}\,[\![b]\!],\ \mathcal{C}\,[\![c]\!]\,(\mathcal{C}\,[\![\texttt{while (}b\texttt{) do }c]\!]\,s\;S)\;S,\ s)$$

  Es ist dabei implizit von den Parametern s und S abhängig: Sein kleinster Fixpunkt definiert $\mathcal{C}\,[\![\texttt{while (}b\texttt{) do }c]\!]\,s\;S$ nur für feste s und S.

Wie in Kap. 7.3 muss man nun noch nachweisen, dass FIX wirklich definiert ist. Dies ist diesmal aber aufwändiger, weil der Parameter f des Funktionals der Semantikfunktion $\mathcal{C}\,[\![c]\!]$ des Schleifenrumpfs c als Parameter übergeben wird, d.h., Monotonie und Kettenstetigkeit des Funktionals erfordern, dass $\mathcal{C}\,[\![c]\!]$ selbst *monoton* und *kettenstetig* ist. Dafür benötigen wir einige Vorbereitungen:

Lemma 120 (Diagonalregel). Sei $(D, \sqsubseteq_D)$ eine Halbordnung, $(E, \sqsubseteq_E)$ eine ccpo und sei $f :: D \Rightarrow D \Rightarrow E$ monoton in beiden Parametern. Dann gilt für alle Ketten Y:

$$\bigsqcup\nolimits_E \{ \bigsqcup\nolimits_E \{ f(d_1)(d_2) \mid d_2 \in Y \} \mid d_1 \in Y \} = \bigsqcup\nolimits_E \{ f(d)(d) \mid d \in Y \}$$

Beweis.

- Alle vorkommenden $\bigsqcup$ existieren: Da f monoton ist, zeigt man leicht, dass alle vorkommenden Mengen Ketten sind. Da E eine ccpo ist, existieren auch die kleinsten oberen Schranken.
- $\sqsubseteq_E$: Es genügt zu zeigen, dass die rechte Seite eine obere Schranke von $\{ \bigsqcup_E \{ f(d_1)(d_2) \mid d_2 \in Y \} \mid d_1 \in Y \}$ ist. Dafür genügt es zu zeigen, dass für alle $d_1 \in Y$ die rechte Seite eine obere Schranke von $\{ f(d_1)(d_2) \mid d_2 \in Y \}$ ist.

  Seien also $d_1, d_2 \in Y$. Da Y eine Kette ist, gilt $d_1 \sqsubseteq_D d_2$ oder $d_2 \sqsubseteq_D d_1$. ObdA. gelte $d_1 \sqsubseteq_D d_2$ – der andere Fall ist symmetrisch. Dann gilt: $f(d_1)(d_2) \sqsubseteq_E f(d_2)(d_2) \sqsubseteq_E \bigsqcup_E \{ f(d)(d) \mid d \in Y \}$.
- $\sqsupseteq_E$: Es genügt zu zeigen, dass die linke Seite eine obere Schranke von $\{ f(d)(d) \mid d \in Y \}$ ist.

  Sei also $d \in Y$. Dann gilt:

  $$f(d)(d) \sqsubseteq_E \bigsqcup\nolimits_E \{ f(d)(d_2) \mid d_2 \in Y \} \sqsubseteq_E \bigsqcup\nolimits_E \{ \bigsqcup\nolimits_E \{ f(d_1)(d_2) \mid d_2 \in Y \} \mid d_1 \in Y \}$$ □

Die Diagonalregel erlaubt, mehrere obere Schranken über die gleiche Kette zu einer oberen Schranke zusammenzufassen.

Definition 121 (Punktweise Halbordnung). Sei $(D, \sqsubseteq)$ eine Halbordnung. Die punktweise Halbordnung $\sqsubseteq'$ auf $A \Rightarrow D$ ist definiert durch $f \sqsubseteq' g$ gdw. $\forall a \in A.\ f(a) \sqsubseteq g(a)$

Lemma 122. $(A \Rightarrow D, \sqsubseteq')$ ist eine Halbordnung. Wenn $(D, \sqsubseteq)$ eine ccpo ist, so ist $(A \Rightarrow D, \sqsubseteq')$ auch eine ccpo mit der kleinsten oberen Schranke

$$\left(\bigsqcup{}' Y\right)(n) = \bigsqcup \{\, f(n) \mid f \in Y \,\}$$

Beweis. Der Beweis, dass $(A \Rightarrow D, \sqsubseteq')$ ist eine Halbordnung, ist trivial.

Sei also $(D, \sqsubseteq)$ eine ccpo und Y eine Kette in $(A \Rightarrow D, \sqsubseteq')$.

- $\bigsqcup' Y$ ist wohldefiniert: $Y(a) = \{\, f(a) \mid f \in Y \,\}$ ist eine Kette in $(D, \sqsubseteq)$.

  Sei $d_1, d_2 \in Y(a)$. Dann gibt es $f_1, f_2 \in Y$ mit $f_1(a) = d_1$ und $f_2(a) = d_2$. Da Y eine Kette ist, gilt $f_1 \sqsubseteq' f_2$ oder $f_2 \sqsubseteq' f_1$. Damit nach Definition aber $f_1(a) \sqsubseteq' f_2(a)$ oder $f_2(a) \sqsubseteq' f_1(a)$.
- $\bigsqcup' Y$ ist eine obere Schranke von Y: Sei $f \in Y$. Zu zeigen: $f \sqsubseteq' \bigsqcup' Y$.

  Nach Definition ist für beliebiges $a \in A$ zu zeigen, dass $f(a) \sqsubseteq \left(\bigsqcup' Y\right)(a)$. Wegen $f \in Y$ ist $f(a) \in Y(a)$. Damit also $f(a) \sqsubseteq \bigsqcup (Y(a)) = \left(\bigsqcup' Y\right)(a)$.
- $\bigsqcup' Y$ ist die kleinste obere Schranke von Y: Sei f obere Schranke von Y. Zu zeigen: $\bigsqcup' Y \sqsubseteq' f$.

  Nach Definition ist für alle $a \in A$ zu zeigen, dass $\left(\bigsqcup' Y\right)(a) \sqsubseteq f(a)$. Wegen $\left(\bigsqcup' Y\right)(a) = \bigsqcup (Y(a))$ genügt es zu zeigen, dass $f(a)$ obere Schranke von $Y(a)$ ist.

  Sei $d \in Y(a)$ beliebig. Dann gibt es ein $f_0 \in Y$ mit $f_0(a) = d$. Da f obere Schranke von Y ist, gilt $f_0 \sqsubseteq' f$, also $d = f_0(a) \sqsubseteq f(a)$ nach Definition von $\sqsubseteq'$. □

In Fall der Ausnahmefortsetzungsumgebungen verwenden wir die Ordnung $\sqsubseteq'$ für $A = \mathsf{Xcp}$ und $D = \mathsf{Cont}$.

Lemma 123 (Monotonie und Kettenstetigkeit der Ausnahmeumgebungsaktualisierung). Die Funktion $f(S)(s) = S[X \mapsto s]$ für festes X ist monoton und kettenstetig in beiden Argumenten.

Beweis. Nachrechnen! □

Lemma 124. Seien $(D, \sqsubseteq_D)$, $(E, \sqsubseteq_E)$ und $(F, \sqsubseteq_F)$ ccpos. Seien $f :: D \Rightarrow E \Rightarrow F$ und $g :: D \Rightarrow E$ monoton in allen Argumenten. Dann ist auch $h(d) = f(d)(g(d))$ monoton. Sind f und g zusätzlich kettenstetig in allen Argumenten, so ist auch h kettenstetig.

Beweis. Monotonie: Sei $d_1 \sqsubseteq_D d_2$. Dann gilt $g(d_1) \sqsubseteq_E g(d_2)$ wegen der Monotonie von g. Mit der Monotonie von f folgt dann: $h(d_1) = f(d_1)(g(d_1)) \sqsubseteq_F f(d_2)(g(d_1)) \sqsubseteq_F f(d_2)(g(d_2)) = h(d_2)$.

Kettenstetigkeit: Sei Y nicht-leere Kette in D. Dann gilt wegen der Kettenstetigkeit von f und g:

$$\begin{aligned} h(\bigsqcup Y) &= f(\bigsqcup Y)(g(\bigsqcup Y)) = \bigsqcup \{\, f(d)(g(\bigsqcup Y)) \mid d \in Y \,\} \\ &= \bigsqcup \{\, f(d)(\bigsqcup \{\, g(d') \mid d' \in Y \,\}) \mid d \in Y \,\} = \bigsqcup \{\, \bigsqcup \{\, f(d)(g(d')) \mid d' \in Y \,\} \mid d \in Y \,\} = \ldots \end{aligned}$$

Die Funktion $h'(d)(d') = f(d)(g(d'))$ ist monoton, also folgt mit der Diagonalregel (Lem. 120):

$$\ldots = \bigsqcup \{\, f(d)(g(d)) \mid d \in Y \,\} = \bigsqcup \{\, h(d) \mid d \in Y \,\}$$ □

Korollar 125. Beliebige Kombinationen monotoner bzw. kettenstetiger Funktionen durch Funktionsanwendung sind monoton bzw. kettenstetig.

Beweis. Kombiniert man kettenstetiger Funktionen nur durch Funktionsanwendung und Parameter, so kann man diese Kombination aus mit den SKI-Kombinatoren der kombinatorischen Logik schreiben. Lem. 124 zeigt, dass der S-Kombinator kettenstetig ist. Die Kombinatoren K (konstante Funktion) und I (Identität) sind trivialerweise kettenstetig. □

Lem. 124 und Kor. 125 verallgemeinern Lem. 103 und Lem. 107 auf beliebige Kompositionen monotoner bzw. kettenstetiger Funktionen.

Lemma 126 (FIX ist monoton). Sei $(D, \sqsubseteq)$ eine ccpo und seien $f_1 \sqsubseteq' f_2$ monoton und kettenstetig. Dann ist $\mathrm{FIX}\,(f_1) \sqsubseteq \mathrm{FIX}\,(f_2)$.

Beweis. Wegen $\mathrm{FIX}\,(f_1) = \bigsqcup \{\, f_1^n(\bot) \mid n \in \mathbb{N} \,\}$ genügt es zu zeigen, dass $\mathrm{FIX}\,(f_2)$ obere Schranke von $\{\, f_1^n(\bot) \mid n \in \mathbb{N} \,\}$ ist. Sei also $n \in \mathbb{N}$. Mittels Induktion über n erhält man $f_1^n(\bot) \sqsubseteq f_2^n(\bot)$:

- Fall n = 0: $f_1^0(\bot) = \bot \sqsubseteq \bot = f_2^0(\bot)$.
- Fall n + 1: Induktionsannahme: $f_1^n(\bot) \sqsubseteq f_2^n(\bot)$.
  Zusammen mit der Monotonie von f_1 und $f_1 \sqsubseteq' f_2$ gilt:

$$f_1^{n+1}(\bot) = f_1(f_1^n(\bot)) \sqsubseteq f_1(f_2^n(\bot)) \sqsubseteq f_2(f_2^n(\bot)) = f_2^{n+1}(\bot)$$

Wegen $\mathrm{FIX}\,(f_2) = \bigsqcup \{\, f_2^n(\bot) \mid n \in \mathbb{N} \,\}$ gilt also $f_1^n(\bot) \sqsubseteq f_2^n(\bot) \sqsubseteq \mathrm{FIX}\,(f_2)$, d. h., $\mathrm{FIX}\,(f_2)$ ist obere Schranke und $\mathrm{FIX}\,(f_1)$ als kleinste obere Schranke ist kleiner oder gleich $\mathrm{FIX}\,(f_2)$. □

Lemma 127 (FIX ist kettenstetig). Sei $(D, \sqsubseteq)$ eine ccpo und Z eine nicht-leere Kette in $(D \Rightarrow D, \sqsubseteq')$, die nur monotone und kettenstetige Funktionen enthält. Dann ist $\{\, \mathrm{FIX}\,(f) \mid f \in Z \,\}$ eine nicht-leere Kette in $(D, \sqsubseteq)$, $\bigsqcup' Z$ ist monoton und kettenstetig, und es gilt:

$$\mathrm{FIX}\left(\bigsqcup' Z\right) = \bigsqcup \{\, \mathrm{FIX}\,(f) \mid f \in Z \,\}$$

Beweis.

- $\{\, \mathrm{FIX}\,(f) \mid f \in Z \,\}$ ist nicht-leere Kette: Da FIX monoton ist und alle $f \in Z$ monoton und kettenstetig sind (Lem. 126), folgt dies aus der Kettenerhaltung (Lem. 104).
- $\bigsqcup' Z$ ist monoton: Sei $d_1 \sqsubseteq d_2$. Wegen $\left(\bigsqcup' Z\right)(d_1) = \bigsqcup \{\, f(d_1) \mid f \in Z \,\}$ genügt es zu zeigen, dass $\left(\bigsqcup' Z\right)(d_2)$ eine obere Schranke von $\{\, f(d_1) \mid f \in Z \,\}$ ist, da $\left(\bigsqcup' Z\right)(d_1)$ die kleinste solche ist.

  Sei also $f \in Z$. Da f monoton ist, gilt $f(d_1) \sqsubseteq f(d_2) \sqsubseteq \bigsqcup \{\, f(d_2) \mid f \in Z \,\} = \left(\bigsqcup' Z\right)(d_2)$.
- $\bigsqcup' Z$ ist kettenstetig: Sei Y nicht-leere Kette in $(D, \sqsubseteq)$. Nach Lem. 104 genügt es zu zeigen, dass $\left(\bigsqcup' Z\right)\left(\bigsqcup Y\right) \sqsubseteq \bigsqcup \{\, \left(\bigsqcup' Z\right)(d) \mid d \in Y \,\}$.

  Da alle $f \in Z$ kettenstetig sind, gilt

$$\begin{aligned}\left(\bigsqcup' Z\right)\left(\bigsqcup Y\right) &= \bigsqcup \{\, f(\bigsqcup Y) \mid f \in Z \,\} = \bigsqcup \{ \bigsqcup \{\, f(d) \mid d \in Y \,\} \mid f \in Z \,\} \\ &\sqsubseteq \bigsqcup \{ \bigsqcup \{\, f(d) \mid f \in Z \,\} \mid d \in Y \,\} = \bigsqcup \{\, (\bigsqcup' Z)(d) \mid d \in Y \,\}\end{aligned}$$

  wobei die Vertauschung der Limiten im $\sqsubseteq$-Schritt einfach nachzurechnen ist.
- FIX ist kettenstetig:
  Wir zeigen die Gleichheit über Antisymmetrie:
  - $\sqsubseteq$: Da $\mathrm{FIX}\left(\bigsqcup' Z\right)$ kleinster Fixpunkt von $\bigsqcup' Z$ ist, genügt es zu zeigen, dass die rechte Seite ein Fixpunkt von $\bigsqcup' Z$ ist.

    Da alle $g \in Z$ kettenstetig sind, gilt:

$$\begin{aligned}\left(\bigsqcup' Z\right)\left(\bigsqcup \{\, \mathrm{FIX}\,(f) \mid f \in Z \,\}\right) &= \bigsqcup \left\{\, g\left(\bigsqcup \{\, \mathrm{FIX}\,(f) \mid f \in Z \,\}\right) \mid g \in Z \,\right\} \\ &= \bigsqcup \left\{ \bigsqcup \{\, g(\mathrm{FIX}\,(f)) \mid f \in Z \,\} \mid g \in Z \,\right\} = \ldots\end{aligned}$$

    Da FIX monoton ist (Lem. 126), lässt sich die Diagonalregel anwenden, um die Schranken zusammenzufassen:

$$\ldots = \bigsqcup \{\, f(\mathrm{FIX}\,(f)) \mid f \in Z \,\} = \bigsqcup \{\, \mathrm{FIX}\,(f) \mid f \in Z \,\}$$

– $\sqsupseteq$: Hier genügt es zu zeigen, dass $\mathrm{FIX}\left(\bigsqcup' Z\right)$ eine obere Schranke von $\{\,\mathrm{FIX}\,(f) \mid f \in Z\,\}$ ist. Sei also $f \in Z$. Da $f \sqsubseteq' \bigsqcup' Z$ und f und $\bigsqcup' Z$ monoton und kettenstetig sind, folgt aus Lem. 126, dass $\mathrm{FIX}\,(f) \sqsubseteq \mathrm{FIX}\left(\bigsqcup' Z\right)$. □

Jetzt haben wir das Werkzeug, um die Wohldefiniertheit von $\mathcal{C}\,[\![\_]\!]$ zu zeigen.

Theorem 128. $\mathcal{C}\,[\![c]\!]\ f\ S$ ist wohldefiniert, monoton und kettenstetig in f und S.

Beweis. Induktion über c.

- Fall $c = \texttt{skip}$: Die Funktion $\lambda f.\ \mathcal{C}\,[\![\texttt{skip}]\!]\ f\ S$ ist die Identität, die trivialerweise monoton und kettenstetig ist. Ebenso ist $\lambda S.\ \mathcal{C}\,[\![\texttt{skip}]\!]\ f\ S$ als konstante Funktion monoton und kettenstetig.
- Fall $c = x\ \texttt{:=}\ a$: Es gilt $\mathcal{C}\,[\![x\ \texttt{:=}\ a]\!]\ f\ S$ ist konstant in S, also trivialerweise monoton und kettenstetig in S. Da $\mathcal{C}\,[\![x\ \texttt{:=}\ a]\!]\ f\ S = (\lambda f.\ f \circ (\lambda\sigma.\ \sigma[x \mapsto \mathcal{A}\,[\![a]\!]\,\sigma]))\ f$ haben wir die Monotonie und Kettenstetigkeit bereits im Beweis von Thm. 109 gezeigt.
- Fall $c = c_1\,;\ c_2$: In f: $\mathcal{C}\,[\![c_1\,;\ c_2]\!]\ f\ S = ((\lambda f.\ \mathcal{C}\,[\![c_1]\!]\ f\ S) \circ (\lambda f.\ \mathcal{C}\,[\![c_2]\!]\ f\ S))\ f$ ist die Hintereinanderausführung der Funktionen $\lambda f.\ \mathcal{C}\,[\![c_1]\!]\ f\ S$ und $\lambda f.\ \mathcal{C}\,[\![c_2]\!]\ f\ S$, die nach Induktionsannahme monoton und kettenstetig sind. Damit ist auch ihre Hintereinanderausführung monoton und kettenstetig.

  In S: Wegen $\mathcal{C}\,[\![c_1\,;\ c_2]\!]\ f\ S = (\lambda S f.\ \mathcal{C}\,[\![c_1]\!]\ f\ S)\ S\ (\mathcal{C}\,[\![c_2]\!]\ f\ S)$ folgt Monotonie und Kettenstetigkeit direkt durch Anwendung von Lem. 124, dessen Annahmen genau die Induktionsannahmen sind.
- Fall $c = \texttt{if (}b\texttt{) then}\ c_1\ \texttt{else}\ c_2$: Auch $\mathcal{C}\,[\![\texttt{if (}b\texttt{) then}\ c_1\ \texttt{else}\ c_2]\!]$ ist nur eine Kombination kettenstetiger Funktionen: $\lambda f\ g.\ \mathrm{IF}\,(\mathcal{B}\,[\![b]\!],\ f,\ g)$ ist monoton und kettenstetig in f und g, $\mathcal{C}\,[\![c_1]\!]$ und $\mathcal{C}\,[\![c_2]\!]$ sind nach Induktionsannahme monoton und kettenstetig.
- Fall $c = \texttt{while (}b\texttt{) do}\ c'$:
  Induktionsannahme: $\mathcal{C}\,[\![c']\!]$ ist monoton und kettenstetig in beiden Argumenten.

  Bezeichne mit $F(f)(S) = \lambda g.\ \mathrm{IF}\,(\mathcal{B}\,[\![b]\!],\ \mathcal{C}\,[\![c']\!]\ g\ S,\ f)$ das Funktional.
  – Wohldefiniertheit: Das Funktional $F(f)(S)$ ist für beliebige f und S monoton und kettenstetig. Wie schon bei der direkten denotationalen Semantik für While lässt sich $F(f)(S)$ als Hintereinanderausführung schreiben: $F(f)(S) = (\lambda g.\ \mathrm{IF}\,(\mathcal{B}\,[\![b]\!],\ g,\ f)) \circ (\lambda g.\ \mathcal{C}\,[\![c']\!]\ g\ S)$. In Kapitel 7.3 haben wir schon gezeigt, dass der vordere Teil monoton und kettenstetig ist, der hintere erfüllt dies laut Induktionsannahme. NB: Für diesen Schritt haben wir den Induktionsbeweis aufgezogen.
  – Monotonie und Kettenstetigkeit: Da das Funktional F monoton und kettenstetig ist, ist FIX auch monoton und kettenstetig (Lem. 126 und 127). Damit haben wir auch hier wieder eine Kombination monotoner und kettenstetiger Funktionen, d.h., $\mathcal{C}\,[\![\texttt{while (}b\texttt{) do}\ c']\!]$ selbst ist auch monoton und kettenstetig.
- Fall $c = \texttt{raise}\ X$: Folgt direkt durch ausrechnen, da Y nicht-leer ist:

$$\mathcal{C}\,[\![\texttt{raise}\ X]\!]\ f_1\ S_1 = S_1(X) \sqsubseteq S_2(X) = \mathcal{C}\,[\![\texttt{raise}\ X]\!]\ f_2\ S_2 \qquad \text{für } f_1 \sqsubseteq f_2 \text{ und } S_1 \sqsubseteq' S_2$$
$$\mathcal{C}\,[\![\texttt{raise}\ X]\!]\ \left(\bigsqcup Y\right)\ S = S(X) = \bigsqcup \{\,S(X) \mid f \in Y\,\} = \bigsqcup \{\mathcal{C}\,[\![\texttt{raise}\ X]\!]\ f\ S \mid f \in Y\,\}$$
$$\mathcal{C}\,[\![\texttt{raise}\ X]\!]\ f\ \left(\bigsqcup' Z\right) = \left(\bigsqcup' Z\right)(X) = \bigsqcup \{\,S(X) \mid S \in Z\,\} = \bigsqcup \{\,\mathcal{C}\,[\![\texttt{raise}\ X]\!]\ f\ S \mid S \in Z\,\}$$

- Fall $c = \texttt{try}\ c_1\ \texttt{catch}\ X\ c_2$: Induktionsannahmen: $\mathcal{C}\,[\![c_1]\!]$ und $\mathcal{C}\,[\![c_2]\!]$ sind monoton und kettenstetig in beiden Argumenten.

  Da $\mathcal{C}\,[\![\texttt{try}\ c_1\ \texttt{catch}\ X\ c_2]\!]\ f\ S = \mathcal{C}\,[\![c_1]\!]\ f\ ((\lambda f.\ S[X \mapsto f])(\mathcal{C}\,[\![c_2]\!]\ f\ S))$, ist auch
  $\mathcal{C}\,[\![\texttt{try}\ c_1\ \texttt{catch}\ X\ c_2]\!]\ f\ S$ nur eine Kombination kettenstetiger Funktionen (Induktionsannahme und Lem. 123), und damit selbst auch monoton und kettenstetig in f.

  Für S teilt man $\mathcal{C}\,[\![\texttt{try}\ c_1\ \texttt{catch}\ X\ c_2]\!]\ f\ S$ in $(\mathcal{C}\,[\![c_1]\!]\ f \circ h)(S)$ auf, wobei
  $h(S) = (\lambda S\ s.\ S[X \mapsto s])(S)(\mathcal{C}\,[\![c_2]\!]\ s\ S)$. h entspricht dann wieder der Form von Lem. 124 und damit ist $\mathcal{C}\,[\![\texttt{try}\ c_1\ \texttt{catch}\ X\ c_2]\!]\ f\ S$ auch monoton und kettenstetig in S. □

8 Axiomatische Semantik

Operationale und denotationale Semantiken legen die Bedeutung eines Programms direkt fest — als Ableitungsbaum, maximale Ableitungsfolge oder partielle Funktion auf den Zuständen. Dies ist eine Art *interner Spezifikation* der Semantik eines Programms: Man beschreibt, was genau die Bedeutungsobjekte sind. Dies ist ein geeigneter Ansatz, wenn man diese Bedeutungsobjekte selbst weiter verwenden möchte – zum Beispiel, um Programmoptimierungen in einem Compiler zu rechtfertigen oder Meta-Eigenschaften der Sprache (wie Typsicherheit) zu beweisen.

Möchte man aber Eigenschaften eines konkreten Programms verifizieren, interessiert man sich nicht für das Bedeutungsobjekt selbst, sondern nur dafür, ob es gewisse Eigenschaften besitzt. Diese Eigenschaften kann man natürlich – wenn die Semantik ausreichende Informationen für die Eigenschaft enthält – von dem Objekt selbst ablesen bzw. beweisen, dies ist aber in vielen Fällen umständlich. Beispielsweise ist man nur an einem Teil der Ergebnisse eines Algorithmus interessiert, also an den Ergebniswerten einzelner Variablen, der Inhalt der anderen (Hilfs-)Variablen ist aber für die spezifische Eigenschaft irrelevant. Der Ableitungsbaum oder die partielle Funktion beinhalten aber auch die gesamte Information über diese Hilfsvariablen, wodurch diese Objekte unnötig viel Information enthalten und damit auch bei einer automatisierten Programmverifikation den Aufwand unnötig erhöhen würden.

Die axiomatische Semantik verfolgt deswegen den Ansatz einer *externen Spezifikation*: Sie legt (mit einem Regelsystem – axiomatisch) fest, welche Eigenschaften das Bedeutungsobjekt jedes Programms haben soll – ohne explizit ein solches Bedeutungsobjekt zu konstruieren. Dadurch werden sämtliche Details einer solchen Konstruktion ausgeblendet (z.B. die Ableitungsfolgen bei der Small-Step- oder die Fixpunktiteration bei der denotationalen Semantik), die für den Nachweis von Programmeigenschaften nur hinderlich sind. Umgekehrt läuft man bei der Erstellung einer axiomatischen Semantik natürlich immer Gefahr, widersprüchliche Bedingungen an die Bedeutungsobjekte zu stellen. Deswegen sollte man zu einer externen Spezifikation immer ein Modell konstruieren, zu einer axiomatischen Semantik also eine operationale oder denotationale finden und die Korrektheit zeigen.

Bei jeder Semantik muss man sich entscheiden, welche Art von Programmeigenschaften man mit ihr ausdrücken können soll. Beispielsweise sind denotationale und Big-Step-Semantik ungeeignet, um die Laufzeit eines Programms, d.h. die Zahl seiner Ausführungsschritte, zu messen. Damit können wir auch keine Eigenschaften über die Laufzeit eines Programms mit diesen Semantiken nachweisen. Zwei wichtige Arten von Eigenschaften lassen sich aber ausdrücken:

Partielle Korrektheit *Falls* das Programm terminiert, dann gilt eine bestimmte Beziehung zwischen Anfangs- und Endzustand.

Totale Korrektheit Das Programm terminiert und es gibt eine bestimmte Beziehung zwischen Anfangs- und Endzustand.

In diesem Kapitel werden wir uns auf partielle Korrektheitsaussagen beschränken.

Beispiel 129. Das niemals terminierende Programm `while (true) do skip` ist korrekt bezüglich allen partiellen Korrektheitseigenschaften. Es ist jedoch für keine solche Beziehung zwischen Anfangs- und Endzustand total korrekt.

8.1 Ein Korrektheitsbeweis mit der denotationalen Semantik

Korrektheitsbeweise von Programmen lassen sich nicht nur mit einer axiomatischer Semantik führen. Auch operationale und denotationale Semantik sind dafür theoretisch vollkommen ausreichend. Dies

wollen wir in diesem Abschnitt am Beispiel der partiellen Korrektheit der Fakultät über die denotationale Semantik vorführen: Sei

$$P = \texttt{y := 1; while (not (x == 1)) do (y := y * x; x := x - 1)}$$

Wir wollen zeigen, dass dieses Programm – sofern es terminiert – in y die Fakultät des Anfangswertes in x speichert. Dies lässt sich als eine Eigenschaft $\varphi(f)$ auf den Bedeutungsobjekten f aus $\Sigma \rightharpoonup \Sigma$ formulieren:

$$\varphi(f) = (\forall \sigma\ \sigma'.\ f(\sigma) = \sigma' \implies \sigma'(\texttt{y}) = (\sigma(\texttt{x}))! \wedge \sigma(\texttt{x}) > 0)$$

P ist also korrekt, wenn $\varphi(\mathcal{D}[\![P]\!])$ gilt. Da $\mathcal{D}[\![P]\!]$ einen Fixpunktoperator enthält, brauchen wir, um dies zu zeigen, noch das Konzept der Fixpunktinduktion.

Definition 130 (Zulässiges Prädikat). Sei $(D, \sqsubseteq)$ eine ccpo. Ein Prädikat $\varphi :: D \Rightarrow \mathbb{B}$ heißt *zulässig*, wenn für alle Ketten Y in $(D, \sqsubseteq)$ gilt: Wenn $\varphi(d) = \mathbf{tt}$ für alle $d \in Y$ gilt, dann auch $\varphi(\bigsqcup Y) = \mathbf{tt}$.

Zulässige Prädikate sind also stetig auf Ketten, auf denen sie überall gelten. Für zulässige Prädikate gibt es folgendes Induktionsprinzip:

Theorem 131 (Fixpunktinduktion, Scott-Induktion). Sei $(D, \sqsubseteq)$ eine ccpo, $f :: D \Rightarrow D$ eine monotone und kettenstetige Funktion, und sei $\varphi :: D \Rightarrow \mathbb{B}$ zulässig. Wenn für alle $d \in D$ gilt, dass aus $\varphi(d) = \mathbf{tt}$ bereits $\varphi(f(d)) = \mathbf{tt}$ folgt, dann gilt auch $\varphi(\mathrm{FIX}\,(f)) = \mathbf{tt}$.

Beweis. Nach Thm. 108 gilt $\mathrm{FIX}\,(f) = \bigsqcup \{\, f^n(\bot) \mid n \in \mathbb{N} \,\}$. Da φ zulässig ist, genügt es also zu zeigen, dass $\varphi(f^n(\bot))$ für alle $n \in \mathbb{N}$ gilt. Beweis durch Induktion über n:

- Basisfall $n = 0$: Nach Def. 130 ($\emptyset$ ist eine Kette) gilt:

$$\varphi(f^0(\bot)) = \varphi(\bot) = \varphi\left(\bigsqcup \emptyset\right) = \mathbf{tt}$$

- Induktionsschritt $n + 1$: Induktionsannahme: $\varphi(f^n(\bot)) = \mathbf{tt}$. Zu zeigen: $\varphi(f^{n+1}(\bot)) = \mathbf{tt}$.
  Aus der Induktionsannahme folgt nach Voraussetzung, dass $\mathbf{tt} = \varphi(f(f^n(\bot))) = \varphi(f^{n+1}(\bot))$. □

Nun zurück zu unserem Beispiel. Mit Thm. 131 können wir jetzt $\varphi(\mathcal{D}[\![P]\!])$ beweisen. Nach Definition gilt:

$$\mathcal{D}[\![P]\!]\,\sigma = \mathcal{D}[\![\texttt{while (not (x == 1)) do (y := y * x; x := x - 1)}]\!]\,(\sigma[\texttt{y} \mapsto 1])$$

und damit

$$\begin{aligned}\varphi(\mathcal{D}[\![P]\!]) &= \varphi(\lambda\sigma.\ \mathcal{D}[\![\texttt{while (not (x == 1)) do (y := y * x; x := x - 1)}]\!]\,(\sigma[\texttt{y} \mapsto 1])) \\ &= \varphi(\lambda\sigma.\ \mathrm{FIX}\,(F)\,(\sigma[\texttt{y} \mapsto 1]))\end{aligned}$$

wobei $F(g) = \mathrm{IF}\,(\mathcal{B}[\![\texttt{not (x == 1)}]\!],\ g \circ \mathcal{D}[\![\texttt{y := y * x; x := x - 1}]\!],\ id)$.

Für die Fixpunkt-Induktion ist φ jedoch zu schwach, da φ nicht unter dem Funktional F erhalten bleibt. Wir brauchen dafür also eine stärkere Eigenschaft φ':

$$\varphi'(f) = (\forall \sigma\ \sigma'.\ f(\sigma) = \sigma' \implies \sigma'(\texttt{y}) = \sigma(\texttt{y}) \cdot (\sigma(\texttt{x}))! \wedge \sigma(\texttt{x}) > 0)$$

Wenn $\varphi'(\mathrm{FIX}\,(F))$ gilt, dann gilt auch $\varphi(\lambda\sigma.\ \mathrm{FIX}\,(F)\,(\sigma[\texttt{y} \mapsto 1]))$. Für die Anwendung der Fixpunktinduktion (Thm. 131) auf φ' und F müssen wir noch folgendes zeigen:

- φ' ist zulässig (Induktionsanfang):
  Sei Y beliebige Kette in $(\Sigma \rightharpoonup \Sigma, \sqsubseteq)$ mit $\varphi'(f) = \mathbf{tt}$ für alle $f \in Y$. Zu zeigen: $\varphi'(\bigsqcup Y) = \mathbf{tt}$.

  Seien also σ, σ' beliebig mit $(\bigsqcup Y)\,\sigma = \sigma'$. Dann gibt es nach Definition ein $f \in Y$ mit $f(\sigma) = \sigma'$. Da $\varphi'(f) = \mathbf{tt}$, gilt $\sigma'(\texttt{y}) = \sigma(\texttt{y}) \cdot (\sigma(\texttt{x}))! \wedge \sigma(\texttt{x}) > 0$, was zu zeigen ist.

- Wenn $\varphi'(f) = \mathbf{tt}$, dann auch $\varphi'(F(f))$ (Induktionsschritt):
  Seien also σ, σ' mit $F(f)(\sigma) = \sigma'$. Zu zeigen: $\sigma'(\mathtt{y}) = \sigma(\mathtt{y}) \cdot (\sigma(\mathtt{x}))!$.
  Beweis durch Fallunterscheidung über $\mathcal{B}[\![\texttt{not (x == 1)}]\!]\sigma$
  - Fall $\mathcal{B}[\![\texttt{not (x == 1)}]\!]\sigma = \mathbf{tt}$: Dann gilt

$$\begin{aligned} F(f)(\sigma) &= (f \circ \mathcal{D}[\![\texttt{y := y * x; x := x - 1}]\!])(\sigma) \\ &= (f \circ \mathcal{D}[\![\texttt{x := x - 1}]\!] \circ \mathcal{D}[\![\texttt{y := y * x}]\!])(\sigma) \\ &= (f \circ \mathcal{D}[\![\texttt{x := x - 1}]\!])(\sigma[\mathtt{y} \mapsto \sigma(\mathtt{y}) \cdot \sigma(\mathtt{x})]) \\ &= f(\sigma[\mathtt{y} \mapsto \sigma(\mathtt{y}) \cdot \sigma(\mathtt{x}),\ \mathtt{x} \mapsto \sigma(\mathtt{x}) - 1]) = \sigma' \end{aligned}$$

    Wegen $\varphi'(f) = \mathbf{tt}$ gilt damit $(\sigma[\mathtt{y} \mapsto \sigma(\mathtt{y}) \cdot \sigma(\mathtt{x}),\ \mathtt{x} \mapsto \sigma(\mathtt{x}) - 1])(\mathtt{x}) > 0$ und damit $\sigma(\mathtt{x}) > 1$, also auch insbesondere $\sigma(\mathtt{x}) > 0$. Außerdem gilt:

$$\begin{aligned} \sigma'(\mathtt{y}) &= (\sigma[\mathtt{y} \mapsto \sigma(\mathtt{y}) \cdot \sigma(\mathtt{x}),\ \mathtt{x} \mapsto \sigma(\mathtt{x}) - 1])(\mathtt{y}) \cdot ((\sigma[\mathtt{y} \mapsto \sigma(\mathtt{y}) \cdot \sigma(\mathtt{x}),\ \mathtt{x} \mapsto \sigma(\mathtt{x}) - 1])(\mathtt{x}))! \\ &= (\sigma(\mathtt{y}) \cdot \sigma(\mathtt{x})) \cdot (\sigma(\mathtt{x}) - 1)! = \sigma(\mathtt{y}) \cdot (\sigma(\mathtt{x}))! \end{aligned}$$

  - Fall $\mathcal{B}[\![\texttt{not (x == 1)}]\!]\sigma = \mathbf{ff}$:
    Wegen $F(f)(\sigma) = id(f)(\sigma) = f(\sigma)$ folgt die Behauptung aus $\varphi'(f)$.

Damit gilt also auch $\varphi'(\mathrm{FIX}\,(F)) = \mathbf{tt}$. Demnach auch $\varphi(\mathcal{D}[\![P]\!])$.

8.2 Zusicherungen

Nach diesem Ausflug in die denotationale Semantik kommen wir nun wirklich zur axiomatischen Beschreibung der Bedeutungsobjekte zurück. Wir konzentrieren uns hierbei auf Aussagen über die partielle Korrektheit eines Programms, die durch Zusicherungen ausgedrückt werden.

Definition 132 (Zusicherung, Hoare-Tripel, Vorbedingung, Nachbedingung). Eine *Zusicherung* (*Hoare-Tripel*) ist ein Tripel $\{\,P\,\}\,c\,\{\,Q\,\}$, wobei das Zustandsprädikat P die *Vorbedingung* und das Zustandsprädikat Q die *Nachbedingung* der Anweisung c ist. Zustandsprädikate sind Funktionen des Typs $\Sigma \Rightarrow \mathbb{B}$.

Eine Zusicherung ist zuerst einmal also nur eine Notation für zwei Prädikate P und Q und eine Anweisung c, der wir im Folgenden noch eine Semantik geben wollen. Intuitiv soll eine Zusicherung $\{\,P\,\}\,c\,\{\,Q\,\}$ aussagen: Wenn das Prädikat P im Anfangszustand σ erfüllt ist, dann wird – sofern die Ausführung von c im Zustand σ terminiert – das Prädikat Q im Endzustand dieser Ausführung erfüllt sein. Terminiert die Ausführung von c im Anfangszustand σ nicht, so macht die Zusicherung keine Aussage.

Beispiel 133. Für das Fakultätsprogramm aus Kap. 8.1 könnte man folgende Zusicherung schreiben, um die Korrektheit auszudrücken.

$$\{\,\mathtt{x} = n\,\}\,\texttt{y := 1; while (not (x == 1)) do (y := y * x; x := x - 1)}\,\{\,\mathtt{y} = n! \wedge n > 0\,\}$$

Dabei ist n eine *logische Variable*, die nicht im Programm vorkommt. Sie wird in der Vorbedingung dazu verwendet, den Anfangswert von x zu speichern, damit er in der Nachbedingung noch verfügbar ist. Würde man stattdessen

$$\{\,\mathbf{tt}\,\}\,\texttt{y := 1; while (not (x == 1)) do (y := y * x; x := x - 1)}\,\{\,\mathtt{y} = \mathtt{x}! \wedge \mathtt{x} > 0\,\}$$

schreiben, hätte dies eine andere Bedeutung: Dann müsste im Endzustand der Wert von y der Fakultät des *Endzustands*werts von x entsprechen. Technisch unterscheiden wir nicht zwischen „echten“ und logischen Variablen, wir speichern beide im Zustand. Da logische Variablen aber nicht im Programm vorkommen, stellen sie keine wirkliche Einschränkung der Programmeigenschaften dar und haben im Endzustand immer noch den gleichen Wert wie am Anfang.

Formal gesehen sind Vor- und Nachbedingungen in Zusicherungen Prädikate auf Zuständen, d.h. vom Typ $\Sigma \Rightarrow \mathbb{B}$. Korrekt hätte die Zusicherung in Beispiel 133 also wie folgt lauten müssen:

$$\{\, \lambda\sigma.\ \sigma(\mathtt{x}) = \sigma(n) \,\} \ldots \{\, \lambda\sigma.\ \sigma(\mathtt{y}) = (\sigma(n))! \wedge \sigma(n) > 0 \,\}$$

Diese umständliche Notation macht aber Zusicherungen nur unnötig schwer lesbar. Innerhalb von Vor- und Nachbedingungen lassen wir deshalb das $\lambda\sigma.$ weg und schreiben nur x statt $\sigma(x)$.

8.3 Inferenzregeln für While

Eine axiomatische Semantik gibt man wie eine Big-Step-Semantik als eine Menge von Inferenzregeln an. Diese Regeln definieren die Ableitbarkeit einer Zusicherung $\{\, P \,\}\, c\, \{\, Q \,\}$, geschrieben als $\vdash \{\, P \,\}\, c\, \{\, Q \,\}$. Dies entspricht einem formalen Beweissystem, mit dem man partiell korrekte Eigenschaften eines Programms nachweisen kann. Für While lauten die Regeln:

$$\textsc{Skip}_\textsc{p}\colon \vdash \{\, P \,\}\, \mathtt{skip}\, \{\, P \,\} \qquad \textsc{Ass}_\textsc{p}\colon \vdash \{\, P[x \mapsto \mathcal{A}\,[\![a]\!]] \,\}\, x\ \mathtt{:=}\ a\, \{\, P \,\}$$

$$\textsc{Seq}_\textsc{p}\colon \frac{\vdash \{\, P \,\}\, c_1\, \{\, Q \,\} \qquad \vdash \{\, Q \,\}\, c_2\, \{\, R \,\}}{\vdash \{\, P \,\}\, c_1\, \mathtt{;}\ c_2\, \{\, R \,\}}$$

$$\textsc{If}_\textsc{p}\colon \frac{\vdash \{\, \lambda\sigma.\ \mathcal{B}\,[\![b]\!]\,\sigma \wedge P(\sigma) \,\}\, c_1\, \{\, Q \,\} \qquad \vdash \{\, \lambda\sigma.\ \neg\mathcal{B}\,[\![b]\!]\,\sigma \wedge P(\sigma) \,\}\, c_2\, \{\, Q \,\}}{\vdash \{\, P \,\}\, \mathtt{if\ (}b\mathtt{)\ then}\ c_1\ \mathtt{else}\ c_2\, \{\, Q \,\}}$$

$$\textsc{While}_\textsc{p}\colon \frac{\vdash \{\, \lambda\sigma.\ \mathcal{B}\,[\![b]\!]\,\sigma \wedge I(\sigma) \,\}\, c\, \{\, I \,\}}{\vdash \{\, I \,\}\, \mathtt{while\ (}b\mathtt{)\ do}\ c\, \{\, \lambda\sigma.\ \neg\mathcal{B}\,[\![b]\!]\,\sigma \wedge I(\sigma) \,\}}$$

$$\textsc{Cons}_\textsc{p}\colon \frac{P \implies P' \qquad \vdash \{\, P' \,\}\, c\, \{\, Q' \,\} \qquad Q' \implies Q}{\vdash \{\, P \,\}\, c\, \{\, Q \,\}}$$

wobei $P[x \mapsto f]$ definiert sei durch

$$(P[x \mapsto f])(\sigma) = P(\sigma[x \mapsto f(\sigma)])$$

und $P \implies P'$ für $\forall\sigma.\ P(\sigma) \implies P'(\sigma)$ steht.

Die Regel für `skip` ist einleuchtend: Was vorher galt, muss auch nachher gelten. Die Regel $\textsc{Ass}_\textsc{p}$ für Zuweisungen x `:=` a nimmt an, dass vor Ausführung im Anfangszustand σ das Prädikat P für den Zustand $\sigma[x \mapsto \mathcal{A}\,[\![a]\!]\,\sigma]$ gilt. Dann muss auch der Endzustand P erfüllen – der in unserer operationalen Semantik eben genau $\sigma[x \mapsto \mathcal{A}\,[\![a]\!]\,\sigma]$ ist.

Neben diesen beiden Axiomen bzw. Axiomschemata des Regelsystems sind die Regeln für die anderen Konstrukte Inferenzregeln, die die Ableitung einer Zusicherung einer zusammengesetzten Anweisung aus den einzelnen Teilen bestimmt. Für die Hintereinanderausführung c_1 `;` c_2 gilt: Die Zusicherung $\{\, P \,\}\, c_1\, \mathtt{;}\ c_2\, \{\, R \,\}$ ist ableitbar, wenn es ein Prädikat Q gibt, das von c_1 unter der Vorbedingung P garantiert wird und unter dessen Voraussetzung c_2 die Nachbedingung R garantieren kann. In der Regel $\textsc{While}_\textsc{p}$ ist I eine Invariante des Schleifenrumpfes, die zu Beginn gelten muss und – falls die Schleife terminiert – auch am Ende noch gilt. Da partielle Korrektheit nur Aussagen über terminierende Ausführungen eines Programms macht, muss an deren Endzustand auch die negierte Schleifenbedingung gelten.

Die letzte Regel, die *Folgeregel* (*rule of consequence*), erlaubt, Vorbedingungen zu verstärken und Nachbedingungen abzuschwächen. Erst damit kann man die anderen Inferenzregeln sinnvoll zusammensetzen.

Beispiel 134. Sei $P = \texttt{if (x == 5) then skip else x := 5}$. Dann gilt:

$$\dfrac{\dfrac{}{\vdash \{\, \mathcal{B}[\![\texttt{x == 5}]\!] \,\}\, \texttt{skip} \,\{\, \texttt{x} = 5 \,\}}\ \textsc{Skip}_\textsc{P} \qquad \dfrac{x \neq 5 \implies \mathbf{tt} \qquad \dfrac{}{\vdash \{\, \mathbf{tt} \,\}\, \texttt{x := 5} \,\{\, \texttt{x} = 5 \,\}}\ \textsc{Ass}_\textsc{P}}{\vdash \{\, \neg\mathcal{B}[\![\texttt{x == 5}]\!] \,\}\, \texttt{x := 5} \,\{\, \texttt{x} = 5 \,\}}\ \textsc{Cons}_\textsc{P}}{\vdash \{\, \mathbf{tt} \,\}\, \texttt{if (x == 5) then skip else x := 5} \,\{\, \texttt{x} = 5 \,\}}\ \textsc{If}_\textsc{P}$$

Beispiel 135. Die Fakultätsberechnung mit einer Schleife (vgl. Kap. 8.1) ist partiell korrekt:

$$\{\, \texttt{x} = n \,\}\, \texttt{y := 1; while (not (x == 1)) do (y := y * x; x := x - 1)} \,\{\, \texttt{y} = n! \wedge n > 0 \,\}$$

$$\dfrac{\dfrac{}{\vdash \{\, \texttt{x} = n \,\}\, \texttt{y := 1} \,\{\, \texttt{x} = n \wedge \texttt{y} = 1 \,\}}\ \textsc{Ass}_\textsc{P} \qquad A}{\vdash \{\, \texttt{x} = n \,\}\, \texttt{y := 1; while (not (x == 1)) do (y := y * x; x := x - 1)} \,\{\, \texttt{y} = n! \wedge n > 0 \,\}}\ \textsc{Seq}_\textsc{P}$$

$$\text{A:}\ \dfrac{\texttt{x} = n \wedge \texttt{y} = 1 \implies I \qquad B \qquad \texttt{x} = 1 \wedge I \implies \texttt{y} = n! \wedge n > 0}{\vdash \{\, \texttt{x} = n \wedge \texttt{y} = 1 \,\}\, \texttt{while (not (x == 1)) do (y := y * x; x := x - 1)} \,\{\, \texttt{y} = n! \wedge n > 0 \,\}}\ \textsc{Cons}_\textsc{P}$$

wobei $I = \texttt{x} \leq 0 \vee (\texttt{y} \cdot \texttt{x}! = n! \wedge \texttt{x} \leq n)$ die Schleifeninvariante ist.

$$\text{B:}\ \dfrac{\dfrac{C \qquad \dfrac{}{\vdash \{\, I[\texttt{x} \mapsto \texttt{x} - 1] \,\}\, \texttt{x := x - 1} \,\{\, I \,\}}\ \textsc{Ass}_\textsc{P}}{\vdash \{\, \mathcal{B}[\![\texttt{not (x == 1)}]\!] \wedge I \,\}\, \texttt{y := y * x; x := x - 1} \,\{\, I \,\}}\ \textsc{Seq}_\textsc{P}}{\vdash \{\, I \,\}\, \texttt{while (not (x == 1)) do (y := y * x; x := x - 1)} \,\{\, \texttt{x} = 1 \wedge I \,\}}\ \textsc{While}_\textsc{P}$$

$$\text{C:}\ \dfrac{D \qquad \dfrac{}{\vdash \{\, (I[\texttt{x} \mapsto \texttt{x} - 1])[\texttt{y} \mapsto \texttt{y} \cdot \texttt{x}] \,\}\, \texttt{y := y * x} \,\{\, I[\texttt{x} \mapsto \texttt{x} - 1] \,\}}\ \textsc{Ass}_\textsc{P}}{\vdash \{\, x \neq 1 \wedge I \,\}\, \texttt{y := y * x} \,\{\, I[\texttt{x} \mapsto \texttt{x} - 1] \,\}}\ \textsc{Cons}_\textsc{P}$$

D: $x \neq 1 \wedge I \implies (I[\texttt{x} \mapsto \texttt{x} - 1])[\texttt{y} \mapsto \texttt{y} \cdot \texttt{x}]$, da:

$$\begin{aligned}
((I[\texttt{x} \mapsto \texttt{x} - 1])[\texttt{y} \mapsto \texttt{y} \cdot \texttt{x}])(\sigma) &= (I[\texttt{x} \mapsto \texttt{x} - 1])(\sigma[\texttt{y} \mapsto \sigma(\texttt{y}) \cdot \sigma(\texttt{x})]) \\
&= I(\sigma[\texttt{y} \mapsto \sigma(\texttt{y}) \cdot \sigma(\texttt{x}),\ \texttt{x} \mapsto (\sigma[\texttt{y} \mapsto \sigma(\texttt{y}) \cdot \sigma(\texttt{x})])(\texttt{x}) - 1]) \\
&= I(\sigma[\texttt{y} \mapsto \sigma(\texttt{y}) \cdot \sigma(\texttt{x}),\ \texttt{x} \mapsto \sigma(\texttt{x}) - 1]) \\
&= \sigma(\texttt{x}) - 1 \leq 0 \vee ((\sigma(\texttt{y}) \cdot \sigma(\texttt{x})) \cdot (\sigma(\texttt{x}) - 1)! = \sigma(n)! \wedge \sigma(\texttt{x}) - 1 \leq \sigma(n)) \\
&= \sigma(\texttt{x}) \leq 1 \vee (\sigma(\texttt{y}) \cdot (\sigma(\texttt{x}))! = \sigma(n)! \wedge \sigma(\texttt{x}) < \sigma(n))
\end{aligned}$$

Bemerkenswert ist, dass für den Korrektheitsbeweis im Beispiel 135 keinerlei Induktion (im Gegensatz zu Kap. 8.1 mit der denotationalen Semantik) gebraucht wurde. Stattdessen musste lediglich für die Ableitung eine Regel nach der anderen angewandt werden – die Essenz des Beweises steckt in der Invariante und in den Implikationen der $\textsc{Cons}_\textsc{P}$-Regel. Damit eignet sich ein solches Beweissystem aus axiomatischen Regeln zur Automatisation: Hat man ein Programm, an dem die Schleifen mit Invarianten annotiert sind, so kann man mit einem *Verifikationsbedingungsgenerator* (*VCG*) automatisch aus den Implikationen der notwendigen $\textsc{Cons}_\textsc{P}$-Regelanwendungen sogenannte Verifikationsbedingungen generieren lassen, die dann bewiesen werden müssen. Da diese Verifikationsbedingungen Prädikate auf *einem* Zustand sind, braucht man sich für deren Lösung nicht mehr um die Programmiersprache, Semantik oder ähnliches kümmern, sondern kann allgemein verwendbare Entscheidungsverfahren anwenden.

8.4 Korrektheit der axiomatischen Semantik

Wie schon in der Einleitung erwähnt, sollte man zeigen, dass das Regelsystem zur Ableitbarkeit von Zusicherungen nicht widersprüchlich ist, d.h., dass es eine operationale oder denotationale Semantik gibt, deren Bedeutungsobjekte die ableitbaren Zusicherungen erfüllen. Gleichbedeutend damit ist, dass man für eine operationale oder denotationale Semantik beweist, dass die Regeln der axiomatischen Semantik korrekt sind: Wenn $\vdash \{ P \}\, c \,\{ Q \}$, dann gilt auch Q auf allen Endzuständen nach Ausführung von c in Startzuständen, die P erfüllen.

Definition 136 (Gültigkeit). Eine Zusicherung $\{ P \}\, c \,\{ Q \}$ ist *gültig* ($\vDash \{ P \}\, c \,\{ Q \}$), wenn für alle σ, σ' mit $\langle c,\ \sigma \rangle \Downarrow \sigma'$ gilt: Aus $P(\sigma)$ folgt $Q(\sigma')$.

Theorem 137 (Korrektheit der axiomatischen Semantik).
Wenn $\vdash \{ P \}\, c \,\{ Q \}$, dann $\vDash \{ P \}\, c \,\{ Q \}$.

Beweis. Beweis durch Regelinduktion über $\vdash \{ P \}\, c \,\{ Q \}$.

- Fall SKIP_P: Zu zeigen: $\vDash \{ P \}\, \texttt{skip} \,\{ P \}$.

  Seien σ, σ' beliebig mit $P(\sigma)$ und $\langle \texttt{skip},\ \sigma \rangle \Downarrow \sigma'$. Mit Regelinversion ($\text{SKIP}_\text{BS}$) auf $\langle \texttt{skip},\ \sigma \rangle \Downarrow \sigma'$ folgt $\sigma' = \sigma$, damit gilt auch $P(\sigma')$, was zu zeigen war.

- Fall ASS_P: Zu zeigen: $\vDash \{ P[x \mapsto \mathcal{A}\,[\![a]\!]] \}\, x \texttt{ := } a \,\{ P \}$.

  Seien σ, σ' beliebig mit $P(\sigma[x \mapsto \mathcal{A}\,[\![a]\!]\,\sigma])$ und $\langle x \texttt{ := } a,\ \sigma \rangle \Downarrow \sigma'$. Zu zeigen: $P(\sigma')$.
  Mit Regelinversion (ASS_BS) folgt $\sigma' = \sigma[x \mapsto \mathcal{A}\,[\![a]\!]\,\sigma]$ und daraus die Behauptung $P(\sigma')$.

- Fall SEQ_P: Induktionsannahmen: $\vDash \{ P \}\, c_1 \,\{ Q \}$ und $\vDash \{ Q \}\, c_2 \,\{ R \}$. Zu zeigen: $\vDash \{ P \}\, c_1 \texttt{;}\ c_2 \,\{ R \}$.

  Seien σ, σ' beliebig mit $P(\sigma)$ und $\langle c_1 \texttt{;}\ c_2,\ \sigma \rangle \Downarrow \sigma'$. Dann gibt es nach Regelinversion (SEQ_BS) ein σ^* mit $\langle c_1,\ \sigma \rangle \Downarrow \sigma^*$ und $\langle c_2,\ \sigma^* \rangle \Downarrow \sigma'$. Aus $\langle c_1,\ \sigma \rangle \Downarrow \sigma^*$ folgt mit der Induktionsannahme $\vDash \{ P \}\, c_1 \,\{ Q \}$ und $P(\sigma)$, dass $Q(\sigma^*)$. Zusammen mit $\langle c_2,\ \sigma^* \rangle \Downarrow \sigma'$ und der Induktionsannahme $\vDash \{ Q \}\, c_2 \,\{ R \}$ folgt $R(\sigma')$, was zu zeigen war.

- Fall IF_P: Induktionsannahmen: $\vDash \{ \mathcal{B}\,[\![b]\!] \wedge P \}\, c_1 \,\{ Q \}$ und $\vDash \{ \neg\mathcal{B}\,[\![b]\!] \wedge P \}\, c_2 \,\{ Q \}$.
  Zu zeigen: $\vDash \{ P \}\, \texttt{if (}b\texttt{) then } c_1 \texttt{ else } c_2 \,\{ Q \}$.

  Seien σ, σ' beliebig mit $P(\sigma)$ und $\langle \texttt{if (}b\texttt{) then } c_1 \texttt{ else } c_2,\ \sigma \rangle \Downarrow \sigma'$. Beweis von $Q(\sigma')$ durch Regelinversion:
  - Fall IFTT_BS: Dann gilt $\mathcal{B}\,[\![b]\!]\,\sigma = \mathbf{tt}$ und $\langle c_1,\ \sigma \rangle \Downarrow \sigma'$. Wegen $P(\sigma)$ gilt auch $(\mathcal{B}\,[\![b]\!] \wedge P)(\sigma)$ und mit der Induktionsannahme $\vDash \{ \mathcal{B}\,[\![b]\!] \wedge P \}\, c_1 \,\{ Q \}$ folgt aus $\langle c_1,\ \sigma \rangle \Downarrow \sigma'$, dass $Q(\sigma')$.
  - Fall IFFF_BS: Analog mit der Induktionsannahme $\vDash \{ \neg\mathcal{B}\,[\![b]\!] \wedge P \}\, c_2 \,\{ Q \}$.

- Fall WHILE_P: Induktionsannahme I: $\vDash \{ \mathcal{B}\,[\![b]\!] \wedge I \}\, c \,\{ I \}$.
  Zu zeigen: $\vDash \{ I \}\, \texttt{while (}b\texttt{) do } c \,\{ \neg\mathcal{B}\,[\![b]\!] \wedge I \}$.

  Seien σ, σ' beliebig mit $\langle \texttt{while (}b\texttt{) do } c,\ \sigma \rangle \Downarrow \sigma'$. Zu zeigen: Wenn $I(\sigma)$, dann $\mathcal{B}\,[\![b]\!]\,\sigma' = \mathbf{ff}$ und $I(\sigma')$. Beweis durch Induktion über $\langle \texttt{while (}b\texttt{) do } c,\ \sigma \rangle \Downarrow \sigma'$:
  - Fall WHILEFF_BS: Dann $\sigma' = \sigma$ und $\mathcal{B}\,[\![b]\!]\,\sigma = \mathbf{ff}$. Daraus folgt direkt die Behauptung.
  - Fall WHILETT_BS: Induktionsannahme II: $\mathcal{B}\,[\![b]\!]\,\sigma = \mathbf{tt}$, $\langle c,\ \sigma \rangle \Downarrow \sigma^*$, und wenn $I(\sigma^*)$, dann $\mathcal{B}\,[\![b]\!]\,\sigma' = \mathbf{ff}$ und $I(\sigma')$. Zu zeigen: Wenn $I(\sigma)$, dann $\mathcal{B}\,[\![b]\!]\,\sigma' = \mathbf{ff}$ und $I(\sigma')$
    Mit der Induktionsannahme II genügt es, zu zeigen, dass aus $I(\sigma)$ auch $I(\sigma^*)$ folgt. Wegen $I(\sigma)$ und $\mathcal{B}\,[\![b]\!]\,\sigma = \mathbf{tt}$ gilt $(\mathcal{B}\,[\![b]\!] \wedge P)(\sigma)$. Mit der Induktionsannahme I folgt aus $\langle c,\ \sigma \rangle \Downarrow \sigma^*$, dass $I(\sigma^*)$.

- Fall CONS_P: Induktionsannahmen: $P \implies P'$, $\vDash \{ P' \}\, c \,\{ Q' \}$ und $Q' \implies Q$.
  Zu zeigen: $\vDash \{ P \}\, c \,\{ Q \}$.

  Seien σ, σ' beliebig mit $P(\sigma)$ und $\langle c,\ \sigma \rangle \Downarrow \sigma'$. Zu zeigen: $Q(\sigma')$.
  Wegen $P \implies P'$ folgt $P'(\sigma)$ aus $P(\sigma)$. Mit $\langle c,\ \sigma \rangle \Downarrow \sigma'$ folgt aus den Induktionsannahmen, dass $Q'(\sigma')$. Wegen $Q' \implies Q$ gilt damit auch $Q(\sigma')$. □

8.5 Vollständigkeit der axiomatischen Semantik

Korrektheit (Thm. 137) sagt aus, dass sich mit den Regeln der axiomatischen Semantik nur Eigenschaften beweisen lassen, die auch in der operationalen gelten. Umgekehrt bedeutet Vollständigkeit eines Kalküls, dass sich alle richtigen Aussagen auch mit den Regeln des Kalküls beweisen lassen. Für die axiomatische Semantik bedeutet dies, dass wenn $\vDash \{ P \}\, c \,\{ Q \}$, dann auch $\vdash \{ P \}\, c \,\{ Q \}$. Diese Vollständigkeit wollen wir in diesem Teil untersuchen.

Definition 138 (Schwächste freie Vorbedingung). Die *schwächste freie Vorbedingung* (*weakest liberal precondition*) $\mathrm{wlp}\,(c,\ Q)$ zu einer Anweisung c und einer Nachbedingung Q ist definiert als

$$\mathrm{wlp}\,(c,\ Q) = \lambda\sigma.\ \forall\sigma'.\ \langle c,\ \sigma\rangle \Downarrow \sigma' \implies Q(\sigma')$$

Sie beschreibt also gerade die Menge von Zuständen, die als Anfangszustand aller terminierenden Ausführungen von c nur zu Endzuständen in Q führen.

Beispiel 139. Die schwächste freie Vorbedingung für $Q = \lambda\sigma.\ \mathbf{ff}$ ist die Menge der Anfangszustände (als Prädikat betrachtet), für die c nicht terminiert. Konkret:

$$\mathrm{wlp}\,(\texttt{while (true) do skip},\ \mathbf{ff})\,\sigma = \mathbf{tt}$$
$$\mathrm{wlp}\,(\texttt{y := 1; while (not (x == 1)) do (y := y * x; x := x - 1)},\ \mathbf{ff})\,\sigma = \sigma(\mathtt{x}) \leq 0$$

Lemma 140. Für alle c und Q ist $\mathrm{wlp}\,(c,\ Q)$ die schwächste mögliche Vorbedingung:

$$\vDash \{ \mathrm{wlp}\,(c,\ Q) \}\, c \,\{ Q \} \qquad \text{und} \qquad \text{wenn, } \vDash \{ P \}\, c \,\{ Q \} \text{ dann } P \implies \mathrm{wlp}\,(c,\ Q)$$

Beweis. Zum Beweis von $\vDash \{ \mathrm{wlp}\,(c,\ Q) \}\, c \,\{ Q \}$ seien σ, σ' beliebig mit $\mathrm{wlp}\,(c,\ Q)\,(\sigma)$ und $\langle c,\ \sigma\rangle \Downarrow \sigma'$. Nach Definition von $\mathrm{wlp}\,(c,\ Q)$ gilt $Q(\sigma')$, was zu zeigen ist.

Sei nun $\vDash \{ P \}\, c \,\{ Q \}$. Zu zeigen: Für alle σ mit $P(\sigma)$ gilt $\mathrm{wlp}\,(c,\ Q)\,(\sigma)$.
Sei also σ' beliebig mit $\langle c,\ \sigma\rangle \Downarrow \sigma'$. Wegen $P(\sigma)$ gilt dann nach $\vDash \{ P \}\, c \,\{ Q \}$ auch $Q(\sigma')$, was zu zeigen ist. □

Lemma 141. Für alle c und Q gilt $\vdash \{ \mathrm{wlp}\,(c,\ Q) \}\, c \,\{ Q \}$.

Beweis. Beweis durch Induktion über c (Q beliebig).

- Fall `skip`: Zu zeigen: $\vdash \{ \mathrm{wlp}\,(\texttt{skip},\ Q) \}\, \texttt{skip} \,\{ Q \}$.

  Es gilt $\mathrm{wlp}\,(\texttt{skip},\ Q)\,(\sigma) = (\forall\sigma'.\, \langle \texttt{skip},\ \sigma\rangle \Downarrow \sigma' \implies Q(\sigma')) = Q(\sigma)$. Damit folgt die Behauptung aus der Regel $\textsc{Skip}_\mathrm{P}$.

- Fall x `:=` a: Zu zeigen: $\vdash \{ \mathrm{wlp}\,(x \texttt{ := } a,\ Q) \}\, x \texttt{ := } a \,\{ Q \}$.

  Es gilt $\mathrm{wlp}\,(x \texttt{ := } a,\ Q) = Q[x \mapsto \mathcal{A}\,[\![a]\!]]$, da

  $$\begin{aligned}\mathrm{wlp}\,(x \texttt{ := } a,\ Q)\,(\sigma) &= (\forall\sigma'.\, \langle x \texttt{ := } a,\ \sigma\rangle \Downarrow \sigma' \implies Q(\sigma')) \\ &= (\forall\sigma'.\ \sigma' = \sigma[x \mapsto \mathcal{A}\,[\![a]\!]\,\sigma] \implies Q(\sigma')) = Q(\sigma[x \mapsto \mathcal{A}\,[\![a]\!]\,\sigma])\end{aligned}$$

  Damit folgt die Behauptung nach der Regel $\textsc{Ass}_\mathrm{P}$.

- Fall $c_1\texttt{;}\ c_2$:
  Induktionsannahmen: Für alle Q gelten $\vdash \{ \mathrm{wlp}\,(c_1,\ Q) \}\, c_1 \,\{ Q \}$ und $\vdash \{ \mathrm{wlp}\,(c_2,\ Q) \}\, c_2 \,\{ Q \}$.
  Zu zeigen: $\vdash \{ \mathrm{wlp}\,(c_1\texttt{;}\ c_2,\ Q) \}\, c_1\texttt{;}\ c_2 \,\{ Q \}$.

  Aus den Induktionsannahmen folgen $\vdash \{ \mathrm{wlp}\,(c_1,\ \mathrm{wlp}\,(c_2,\ Q)) \}\, c_1 \,\{ \mathrm{wlp}\,(c_2,\ Q) \}$ und
  $\vdash \{ \mathrm{wlp}\,(c_2,\ Q) \}\, c_2 \,\{ Q \}$. Damit gilt auch $\vdash \{ \mathrm{wlp}\,(c_1,\ \mathrm{wlp}\,(c_2,\ Q)) \}\, c_1\texttt{;}\ c_2 \,\{ Q \}$ nach Regel $\textsc{Seq}_\mathrm{P}$.
  Daraus folgt die Behauptung nach Regel $\textsc{Cons}_\mathrm{P}$, falls $\mathrm{wlp}\,(c_1\texttt{;}\ c_2,\ Q) \implies \mathrm{wlp}\,(c_1,\ \mathrm{wlp}\,(c_2,\ Q))$.

Für diese Implikation ist für alle σ zu zeigen, dass wenn $\text{wlp}\,(c_1\,;\ c_2,\ Q)\,\sigma$ gilt, dann gilt auch $\text{wlp}\,(c_1,\ \text{wlp}\,(c_2,\ Q))\,\sigma$.
Sei also – nach Definition von $\text{wlp}\,(\_,\ \_)$ – σ' beliebig mit $\langle c_1,\ \sigma\rangle \Downarrow \sigma'$. Zu zeigen: $\text{wlp}\,(c_2,\ Q)\,\sigma'$.
Sei also – wieder nach Definition – σ'' beliebig mit $\langle c_2,\ \sigma'\rangle \Downarrow \sigma''$. Nun bleibt zu zeigen: $Q(\sigma'')$.
Aus den Annahmen $\langle c_1,\ \sigma\rangle \Downarrow \sigma'$ und $\langle c_2,\ \sigma'\rangle \Downarrow \sigma''$ folgt, dass $\langle c_1\,;\ c_2,\ \sigma\rangle \Downarrow \sigma''$. Da $\text{wlp}\,(c_1\,;\ c_2,\ Q)\,\sigma$, folgt die Behauptung $Q(\sigma'')$ nach Definition von $\text{wlp}\,(\_,\ \_)$.

- Fall `if (`b`) then` c_1 `else` c_2:
  Induktionsannahmen: Für alle Q gelten $\vdash \{\,\text{wlp}\,(c_1,\ Q)\,\}\,c_1\,\{\,Q\,\}$ und $\vdash \{\,\text{wlp}\,(c_2,\ Q)\,\}\,c_2\,\{\,Q\,\}$.
  Zu zeigen: $\vdash \{\,\text{wlp}\,(\texttt{if (}b\texttt{) then }c_1\texttt{ else }c_2,\ Q)\,\}\,\texttt{if (}b\texttt{) then }c_1\texttt{ else }c_2\,\{\,Q\,\}$.

  Sei P definiert als

  $$P(\sigma) = (\mathcal{B}\,[\![b]\!]\,\sigma \wedge \text{wlp}\,(c_1,\ Q)\,\sigma) \vee (\neg\mathcal{B}\,[\![b]\!]\,\sigma \wedge \text{wlp}\,(c_2,\ Q)\,\sigma)$$

  Dann gilt mit $c = \texttt{if (}b\texttt{) then }c_1\texttt{ else }c_2$:

  $$\frac{\text{wlp}\,(c,\ Q) \implies P \qquad \dfrac{A \qquad B}{\vdash \{\,P\,\}\,c\,\{\,Q\,\}}\ \textsc{If}_\text{P} \qquad Q \implies Q}{\vdash \{\,\text{wlp}\,(c,\ Q)\,\}\,c\,\{\,Q\,\}}\ \textsc{Cons}_\text{P}$$

  wobei mit den Induktionsannahmen gilt:

  $$\text{A: } \frac{\mathcal{B}\,[\![b]\!] \wedge P \implies \text{wlp}\,(c_1,\ Q) \qquad \vdash \{\,\text{wlp}\,(c_1,\ Q)\,\}\,c_1\,\{\,Q\,\} \qquad Q \implies Q}{\vdash \{\,\mathcal{B}\,[\![b]\!] \wedge P\,\}\,c_1\,\{\,Q\,\}}\ \textsc{Cons}_\text{P}$$

  $$\text{B: } \frac{\neg\mathcal{B}\,[\![b]\!] \wedge P \implies \text{wlp}\,(c_2,\ Q) \qquad \vdash \{\,\text{wlp}\,(c_2,\ Q)\,\}\,c_2\,\{\,Q\,\} \qquad Q \implies Q}{\vdash \{\,\neg\mathcal{B}\,[\![b]\!] \wedge P\,\}\,c_2\,\{\,Q\,\}}\ \textsc{Cons}_\text{P}$$

  Die Implikation $\text{wlp}\,(c,\ Q) \implies P$ wird wie im Fall $c_1\,;\ c_2$ gezeigt.

- Fall `while (`b`) do` c: Induktionsannahme: $\vdash \{\,\text{wlp}\,(c,\ Q)\,\}\,c\,\{\,Q\,\}$ für alle Q
  Zu zeigen: $\vdash \{\,\text{wlp}\,(\texttt{while (}b\texttt{) do }c,\ Q)\,\}\,\texttt{while (}b\texttt{) do }c\,\{\,Q\,\}$.

  Sei $P = \text{wlp}\,(\texttt{while (}b\texttt{) do }c,\ Q)$. Wir wollen zeigen, dass P eine Schleifeninvariante ist. Mit der Induktionsannahme, spezialisiert auf $Q = P$, gilt dann:

  $$\frac{\dfrac{\dfrac{\mathcal{B}\,[\![b]\!] \wedge P \implies \text{wlp}\,(c,\ P) \quad \vdash \{\,\text{wlp}\,(c,\ P)\,\}\,c\,\{\,P\,\}}{\vdash \{\,\mathcal{B}\,[\![b]\!] \wedge P\,\}\,c\,\{\,P\,\}}\ \textsc{Cons}_\text{P}}{\vdash \{\,P\,\}\,\texttt{while (}b\texttt{) do }c\,\{\,\neg\mathcal{B}\,[\![b]\!] \wedge P\,\}}\ \textsc{While}_\text{P} \qquad \neg\mathcal{B}\,[\![b]\!] \wedge P \implies Q}{\vdash \{\,P\,\}\,\texttt{while (}b\texttt{) do }c\,\{\,Q\,\}}\ \textsc{Cons}_\text{P}$$

  Wir müssen dazu aber noch die Implikationen der $\textsc{Cons}_\text{P}$-Anwendungen nachweisen:
  - $\neg\mathcal{B}\,[\![b]\!] \wedge P \implies Q$: Sei σ beliebig mit $\neg\mathcal{B}\,[\![b]\!]\,\sigma \wedge P(\sigma)$, also insbesondere $\mathcal{B}\,[\![b]\!]\,\sigma = \mathbf{ff}$. Dann gilt $\langle\texttt{while (}b\texttt{) do }c,\ \sigma\rangle \Downarrow \sigma$ nach Regel $\textsc{WhileFF}_\text{BS}$. Wegen $P(\sigma)$ folgt damit $Q(\sigma)$ nach Definition.
  - $\mathcal{B}\,[\![b]\!] \wedge P \implies \text{wlp}\,(c,\ P)$: Sei also σ beliebig mit $\mathcal{B}\,[\![b]\!]\,\sigma \wedge P(\sigma)$. Zu zeigen: $\text{wlp}\,(c,\ P)\,\sigma$.
    Sei also σ' beliebig mit $\langle c,\ \sigma\rangle \Downarrow \sigma'$. Zu zeigen: $P(\sigma')$.
    Da $P = \text{wlp}\,(\texttt{while (}b\texttt{) do }c,\ Q)$, sei also σ'' beliebig mit $\langle\texttt{while (}b\texttt{) do }c,\ \sigma'\rangle \Downarrow \sigma''$. Zu zeigen: $Q(\sigma'')$.
    Wegen $\langle c,\ \sigma\rangle \Downarrow \sigma'$ und $\langle\texttt{while (}b\texttt{) do }c,\ \sigma'\rangle \Downarrow \sigma''$ gilt auch $\langle\texttt{while (}b\texttt{) do }c,\ \sigma\rangle \Downarrow \sigma''$ nach Regel $\textsc{WhileTT}_\text{BS}$. Mit $\text{wlp}\,(\texttt{while (}b\texttt{) do }c,\ Q)\,\sigma$ $(= P(\sigma))$, folgt $Q(\sigma'')$. □

Theorem 142 (Vollständigkeit der axiomatischen Semantik).
Wenn $\vDash \{\,P\,\}\,c\,\{\,Q\,\}$, dann $\vdash \{\,P\,\}\,c\,\{\,Q\,\}$.

Beweis. Nach Lem. 140 und 141 gilt:

$$\frac{P \implies \mathrm{wlp}\,(c,\ Q) \qquad \vdash \{\,\mathrm{wlp}\,(c,\ Q)\,\}\,c\,\{\,Q\,\} \qquad Q \implies Q}{\vdash \{\,P\,\}\,c\,\{\,Q\,\}}\ \textsc{Cons}_{\mathrm{P}} \qquad \square$$

Korollar 143. $\vDash \{\,P\,\}\,c\,\{\,Q\,\}$ gdw. $\vdash \{\,P\,\}\,c\,\{\,Q\,\}$.

8.6 Semantische Prädikate und syntaktische Bedingungen

Beispiel 139 suggeriert bereits, dass es keinen Algorithmus geben kann, der die Ableitbarkeit von $\vdash \{\,P\,\}\,c\,\{\,Q\,\}$ entscheiden kann – sonst wäre auch das Halteproblem lösbar: Seien $P = \lambda\sigma.\ \mathbf{tt}$ und $Q = \lambda\sigma.\ \mathbf{ff}$. Die Zusicherung $\vdash \{\,P\,\}\,c\,\{\,Q\,\}$ charakterisiert dann all die Programme c, die niemals anhalten. Ebensowenig ist $\mathrm{wlp}\,(c,\ Q)$ berechenbar. Dies liegt an der Regel $\textsc{Cons}_{\mathrm{P}}$, da Implikationen zwischen beliebigen Prädikaten P und P' nicht entscheidbar sind, man aber auf diese auch nicht verzichten kann. Ein automatisches Verifikationssystem, das alle Programmeigenschaften beweisen kann, ist also auch mit axiomatischer Semantik nicht möglich.

Damit ein solches System überhaupt arbeiten kann, braucht man eine symbolische Darstellung der Prädikate. Dies ist aber nichts anderes als eine Unterscheidung zwischen Syntax und Semantik! Die Menge der Zustandsprädikate ist die Menge der semantischen Bedeutungsobjekte für Vor- und Nachbedingungen – wir haben also bisher nur mit den semantischen Objekten gearbeitet. Jetzt fehlt uns noch die Syntax und die Interpretationsfunktion $\mathcal{I}\,[\![\_]\!]$, die aus den syntaktischen Ausdrücken wieder das semantische Prädikat gewinnt. Genau genommen sind unsere notationellen Vereinfachungen der Zusicherungen aus Kap. 8.2 bereits ein halbherziger Versuch, Syntax für Bedingungen einzuführen.

Wechselt man von semantischen Prädikaten in den Vor- und Nachbedingungen auf syntaktische Bedingungen innerhalb einer solchen Logik, übertragen sich nicht alle Eigenschaften, die wir in diesem Kapitel untersucht haben. Korrektheit (Thm. 137) ist in jedem Fall gewährleistet, sofern die Operationen $\_[\_ \mapsto \_]$, $\lambda\sigma.\ \_(\sigma) \wedge \_(\sigma)$ und $\mathcal{B}\,[\![b]\!]$, die in den Regeln vorkommen, auch in der Syntax semantisch korrekt umgesetzt werden.

Vollständigkeit (Thm. 142) lässt sich dagegen nicht automatisch übertragen: Es ist nicht klar, dass für alle *syntaktischen* Bedingungen Q die schwächste Vorbedingung $\mathrm{wlp}\,(c,\ [\![Q]\!])$ und alle Zwischenbedingungen (z.B. die Schleifeninvarianten), die für die Ableitbarkeit von $\vdash \{\,\mathrm{wlp}\,(c,\ [\![Q]\!])\,\}\,c\,\{\,[\![Q]\!]\,\}$ benötigt werden, in der Logik *syntaktisch ausdrückbar* sind.

Beispiel 144. Sei $c = c_1\,;\ c_2$ mit

$$c_1 = \texttt{while (1 <= x) do (x := x - 1; z := z + y)}$$
$$c_2 = \texttt{while (1 <= z) do z := z} - \texttt{y}$$

Dann gilt $\vDash \{\,\mathtt{z} = 0 \wedge \mathtt{x} \geq 0 \wedge \mathtt{y} \geq 0\,\}\,c\,\{\,\mathtt{z} = 0\,\}$. Die schwächste Vorbedingung zu c_2 und Nachbedingung $\{\,\mathtt{z} = 0\,\}$ ist, dass $\mathtt{z}$ ein Vielfaches von $\mathtt{y}$ ist, d.h., $\exists n.\ \mathtt{z} = \mathtt{y} \cdot n$. Nimmt man Presburger-Arithmetik[6] als Sprache der Zusicherungen, so lässt sich dieser Zusammenhang zwischen $\mathtt{y}$ und $\mathtt{z}$ nicht ausdrücken und damit auch nicht mit dem Kalkül $\vdash \{\,\_\,\}\,\_\,\{\,\_\,\}$ beweisen. Bemerkenswert ist dabei, dass das Programm c selbst gar keine Multiplikation verwendet, sondern diese aus der einfacheren Addition synthetisiert.

Für die mächtigere Sprache „Logik erster Stufe mit Addition und Multiplikation“ (FOL + $(\mathbb{N}, +, \cdot)$) gilt, dass sich alle schwächsten freien Vorbedingungen ausdrücken lassen – diese Eigenschaft heißt *Expressivität*. Damit überträgt sich dann auch die Vollständigkeit (Thm. 142).

[6] Presburger-Arithmetik ist die Logik erster Stufe mit Addition auf den ganzen Zahlen, aber ohne Multiplikation.

Theorem 145. Sei Q eine (syntaktische) Formel aus FOL + $(\mathbb{N}, +, \cdot)$. Dann ist auch $\mathrm{wlp}\,(c,\ Q)$ in FOL + $(\mathbb{N}, +, \cdot)$ ausdrückbar.

Beweisidee.

1. Die Menge V der in Q und c vorkommenden Variablen ist endlich und $\mathrm{wlp}\,(c,\ Q)$ ignoriert die Belegungen aller anderen Variablen. Deswegen lässt sich der relevante Teil eines Zustands als endliches Tupel von Zahlen schreiben. Dieses Tupel lässt sich wiederum in einer einzigen natürlichen Zahl, der Gödelnummer des Zustands, kodieren.
2. Programmausführungen lassen sich als FOL-Formel kodieren. Für ein Programm c mit Variablen aus V gibt es eine FOL-Formel $R_c(\sigma, \sigma')$, so dass $R_c(\sigma, \sigma')$ gilt gdw. $\langle c,\ \sigma\rangle \Downarrow \sigma'$.
3. Dann ist $P(\sigma) = \exists\sigma'.\ R_c(\sigma, \sigma') \wedge Q(\sigma')$ die gesuchte Formel für $\mathrm{wlp}\,(c,\ Q)$. □

Trotz dieses Theorems ist $\vdash \{\ \_\ \}\ \_\ \{\ \_\ \}$ ist nicht rekursiv aufzählbar, d.h., es gibt keinen Algorithmus, der alle wahren Aussagen über alle Programme ausgeben könnte. Ansonsten wäre das Halteproblem entscheidbar, da dann auch $\{\, c \mid \vdash \{\, \texttt{true} \,\}\, c\, \{\, \texttt{false} \,\} \,\}$ rekursiv aufzählbar wäre. Diese Menge besteht aber genau aus den Programmen, die niemals terminieren, und von der ist bekannt, dass sie nicht rekursiv aufzählbar ist. Dies mag zuerst überraschen, weil Thm. 145 eigentlich einen Konstruktionsalgorithmus für die schwächste Vorbedingung für alle Programme liefert. Allerdings darf man mit Regel $\textsc{Cons}_\textsc{p}$ jederzeit Vorbedingungen verschärfen und Implikationen in FOL sind nicht rekursiv aufzählbar.

Deswegen nennt man das Hoare-Kalkül *relativ vollständig*: Es ist vollständig (im Sinne rekursiver Aufzählbarkeit aller Ableitungen) relativ zur Vollständigkeit (d.h., rekursiven Aufzählbarkeit) der verwendeten Formelsprache.

8.7 Verifikationsbedingungen

Axiomatische Semantik eignet sich für den Nachweis konkreter Eigenschaften eines konkreten Programms besser als operationale und denotationale Ansätze, weil die Regeln der axiomatischen Semantik die Programmsyntax in eine Formelsprache überführen, in der die Programmsyntax nicht mehr vorkommt. Der Beweis der Vollständigkeit (Thm. 142) hat gezeigt, dass letztendlich nur die Schleifeninvarianten gefunden und die Anwendungen der Regel $\textsc{Cons}_\textsc{p}$ nachgerechnet werden müssen, der Rest ergibt sich automatisch aus der Programmstruktur. Dies lässt sich automatisieren und liefert einen *Verifikationsbedingungsgenerator* vc.

Im Folgenden bräuchten wir eigentlich eine Formelsyntax für Invarianten, Vor- und Nachbedingungen, aber die konkrete Syntax spielt keine wesentliche Rolle. Es genügt vorläufig, wenn alle booleschen Ausdrücke aus Bexp enthalten sind. Außerdem brauchen wir noch

1. die Interpretationsfunktion $\mathcal{I}\,[\![\_]\!]$ auf Formeln – d.h., $\mathcal{I}\,[\![P]\!]\,\sigma = \mathbf{tt}$ gdw. die Variablenbelegung (= Zustand) σ Formel P erfüllt –, die boolesche Ausdrücke und Konnektoren entsprechend $\mathcal{B}\,[\![\_]\!]$ interpretiert, sowie
2. eine Substitutionsfunktion $\_[x \mapsto a]$, die mit der Interpretationsfunktion kommutiert:
$$\mathcal{I}\,[\![P[x \mapsto a]]\!]\,\sigma = \mathcal{I}\,[\![P]\!]\,(\sigma[x \mapsto \mathcal{A}\,[\![a]\!]\,\sigma])$$

Eine Formel P gilt, wenn sie für alle Belegungen σ wahr ist, d.h., $\mathcal{I}\,[\![P]\!]\,\sigma = \mathbf{tt}$ für alle σ.

In den Beispielen verwenden wir Bexp als Sprache und $\mathcal{B}\,[\![\_]\!]$ als Interpretation. Die Substitutionsfunktionen $a[x \mapsto a']$ bzw. $b[x \mapsto a']$ ersetzen alle Vorkommen von x im arithmetischen bzw. booleschen Ausdruck a bzw. b durch a'. Für sie gilt auch folgendes Substitutionslemma:

Lemma 146 (Substitutionslemma für arithmetische und boolesche Ausdrücke).

$$\mathcal{A}\,[\![a[x \mapsto a']]\!]\,\sigma = \mathcal{A}\,[\![a]\!]\,(\sigma[x \mapsto \mathcal{A}\,[\![a']\!]\,\sigma]) \qquad \text{und} \qquad \mathcal{B}\,[\![b[x \mapsto a']]\!]\,\sigma = \mathcal{B}\,[\![b]\!]\,(\sigma[x \mapsto \mathcal{A}\,[\![a']\!]\,\sigma])$$

Beweis. Induktion über a bzw. b und ausrechnen. □

Definition 147 (Annotiertes Programm). In einem annotierten Programm ist jede Schleife mit einer Schleifeninvariante I annotiert.

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

Die Funktion strip :: ACom ⇒ Com entfernt alle Annotationen eines Programms:

$$\begin{array}{ll}
\text{strip}(\texttt{skip}) & = \texttt{skip} \\
\text{strip}(x \texttt{ := } a) & = x \texttt{ := } a \\
\text{strip}(c_1\texttt{; } c_2) & = \text{strip}(c_1)\texttt{; strip}(c_2) \\
\text{strip}(\texttt{if (}b\texttt{) then } c_1 \texttt{ else } c_2) & = \texttt{if (}b\texttt{) then strip}(c_1)\texttt{ else strip}(c_2) \\
\text{strip}(\texttt{while (}b\texttt{) \{}I\texttt{\} do } c) & = \texttt{while (}b\texttt{) do strip}(c)
\end{array}$$

Die berechnete Vorbedingung $\text{pre}(c, Q)$ für das annotierte Programm c und die Nachbedingung Q ist im Wesentlichen die schwächste freie Vorbedingung – bis auf die annotierten Invarianten:

$$\begin{array}{ll}
\text{pre}(\texttt{skip}, Q) & = Q \\
\text{pre}(x \texttt{ := } a, Q) & = Q[x \mapsto a] \\
\text{pre}(c_1\texttt{; } c_2, Q) & = \text{pre}(c_1, \text{pre}(c_2, Q)) \\
\text{pre}(\texttt{if (}b\texttt{) then } c_1 \texttt{ else } c_2, Q) & = \texttt{(}b\texttt{ -> pre}(c_1, Q)\texttt{) \&\& (not }b\texttt{ -> pre}(c_2, Q)\texttt{)} \\
\text{pre}(\texttt{while (}b\texttt{) \{}I\texttt{\} do } c, Q) & = I
\end{array}$$

wobei b_1 `->` b_2 syntaktischer Zucker für `not` b_1 `||` b_2 ist.

Der Verifikationsbedingungsgenerator generiert als syntaktische Formeln die Implikationen, die bei einer Ableitung von $\vdash \{\, \text{pre}(c, Q) \,\}\, \text{strip}(c)\, \{\, Q \,\}$ in den Schritten mit der Regel ConsP auftreten.

$$\begin{array}{ll}
\text{vc}(\texttt{skip}, Q) & = \texttt{true} \\
\text{vc}(x \texttt{ := } a, Q) & = \texttt{true} \\
\text{vc}(c_1\texttt{; } c_2, Q) & = \text{vc}(c_1, \text{pre}(c_2, Q))\texttt{ \&\& vc}(c_2, Q) \\
\text{vc}(\texttt{if (}b\texttt{) then } c_1 \texttt{ else } c_2, Q) & = \text{vc}(c_1, Q)\texttt{ \&\& vc}(c_2, Q) \\
\text{vc}(\texttt{while (}b\texttt{) \{}I\texttt{\} do } c, Q) & = \texttt{(}\underbrace{b\texttt{ \&\& }I\texttt{ -> pre}(c, I)}_{\text{Schleifeninvariante}}\texttt{) \&\& (}\underbrace{\texttt{not }b\texttt{ \&\& }I\texttt{ -> }Q}_{\text{Schleifenende}}\texttt{) \&\ vc}(c, I)
\end{array}$$

Beispiel 148. Sei

$$w = \texttt{while (}\underbrace{\texttt{not (i == n)}}_{=b}\texttt{) \{}\underbrace{\texttt{2 * x == i * (i + 1)}}_{=I}\texttt{\} do (}\underbrace{\texttt{i := i + 1; x := x + i}}_{=c}\texttt{)}$$

$$Q = \texttt{2 * x == n * (n + 1)}$$

Dann gilt:

$$\begin{array}{rl}
\text{pre}(c, I) & = I[\texttt{x} \mapsto \texttt{x + i}, \texttt{i} \mapsto \texttt{i + 1}] = \texttt{2 * (x + (i + 1)) == (i + 1) * ((i + 1) + 1)} \\
\text{pre}(w, Q) & = I \\
\text{vc}(c, I) & = \texttt{true \&\& true} \\
\text{vc}(w, Q) & = \texttt{(not (i == n) \&\& }I\texttt{ -> pre}(c, I)\texttt{) \&\& (not not (i == n) \&\& }I\texttt{ -> }Q\texttt{) \&\& vc}(c, I)
\end{array}$$

Theorem 149 (Korrektheit des Verifikationsbedingungsgenerators).
Wenn $\mathrm{vc}(c, Q)$ gilt, dann $\vdash \{\, \mathcal{I}\,[\![\mathrm{pre}(c, Q)]\!] \,\}\, \mathrm{strip}(c)\, \{\, \mathcal{I}\,[\![Q]\!] \,\}$.

Beweis. Induktion über c (Q beliebig):

- Fall `skip`: Wegen $\mathrm{pre}(\texttt{skip}, Q) = Q$ und $\mathrm{strip}(\texttt{skip}) = \texttt{skip}$ gilt nach Regel $\textsc{Skip}_\mathrm{P}$:

$$\vdash \{\, \mathcal{I}\,[\![\mathrm{pre}(\texttt{skip}, Q)]\!] \,\}\, \mathrm{strip}(\texttt{skip})\, \{\, \mathcal{I}\,[\![Q]\!] \,\}$$

- Fall x `:=` a: Wegen $\mathrm{pre}(x \texttt{ := } a, Q) = Q[x \mapsto a]$ und $\mathcal{I}\,[\![Q[x \mapsto a]]\!] = \mathcal{I}\,[\![Q]\!]\,[x \mapsto \mathcal{A}\,[\![a]\!]]$ und $\mathrm{strip}(x \texttt{ := } a) = x \texttt{ := } a$ folgt die Behauptung $\vdash \{\, \mathcal{I}\,[\![\mathrm{pre}(x \texttt{ := } a, Q)]\!] \,\}\, \mathrm{strip}(x \texttt{ := } a)\, \{\, \mathcal{I}\,[\![Q]\!] \,\}$ nach Regel $\textsc{Ass}_\mathrm{P}$.

- Fall c_1`;` c_2: Induktionsannahmen (für beliebige Q):
  Wenn $\mathrm{vc}(c_1, Q)$ gilt, dann $\vdash \{\, \mathcal{I}\,[\![\mathrm{pre}(c_1, Q)]\!] \,\}\, \mathrm{strip}(c_1)\, \{\, \mathcal{I}\,[\![Q]\!] \,\}$.
  Wenn $\mathrm{vc}(c_2, Q)$ gilt, dann $\vdash \{\, \mathcal{I}\,[\![\mathrm{pre}(c_2, Q)]\!] \,\}\, \mathrm{strip}(c_2)\, \{\, \mathcal{I}\,[\![Q]\!] \,\}$.

  Sei Q beliebig, so dass $\mathrm{vc}(c_1 \texttt{;}\ c_2, Q)$ gelte. Zu zeigen: $\vdash \{\, \mathcal{I}\,[\![\mathrm{pre}(c_1 \texttt{;}\ c_2, Q)]\!] \,\}\, \mathrm{strip}(c_1 \texttt{;}\ c_2)\, \{\, \mathcal{I}\,[\![Q]\!] \,\}$.

  Wegen $\mathrm{vc}(c_1 \texttt{;}\ c_2, Q) = \mathrm{vc}(c_1, \mathrm{pre}(c_2, Q))$ `&&` $\mathrm{vc}(c_2, Q)$ gelten auch $\mathrm{vc}(c_1, \mathrm{pre}(c_2, Q))$ und $\mathrm{vc}(c_2, Q)$ und damit nach Induktionsannahme

  $$\vdash \{\, \mathcal{I}\,[\![\mathrm{pre}(c_1, \mathrm{pre}(c_2, Q))]\!] \,\}\, \mathrm{strip}(c_1)\, \{\, \mathcal{I}\,[\![\mathrm{pre}(c_2, Q)]\!] \,\} \quad \text{und} \quad \vdash \{\, \mathcal{I}\,[\![\mathrm{pre}(c_2, Q)]\!] \,\}\, \mathrm{strip}(c_2)\, \{\, \mathcal{I}\,[\![Q]\!] \,\}$$

  Daraus folgt nach Regel $\textsc{Seq}_\mathrm{P}$ die Behauptung, da $\mathrm{pre}(c_1 \texttt{;}\ c_2, Q) = \mathrm{pre}(c_1, \mathrm{pre}(c_2, Q))$.

- Fall `if (`b`) then` c_1 `else` c_2: Induktionsannahmen (für beliebige Q):
  Wenn $\mathrm{vc}(c_1, Q)$ gilt, dann $\vdash \{\, \mathcal{I}\,[\![\mathrm{pre}(c_1, Q)]\!] \,\}\, \mathrm{strip}(c_1)\, \{\, \mathcal{I}\,[\![Q]\!] \,\}$.
  Wenn $\mathrm{vc}(c_2, Q)$ gilt, dann $\vdash \{\, \mathcal{I}\,[\![\mathrm{pre}(c_2, Q)]\!] \,\}\, \mathrm{strip}(c_2)\, \{\, \mathcal{I}\,[\![Q]\!] \,\}$.

  Sei Q beliebig, so dass $\mathrm{vc}(\texttt{if (}b\texttt{) then } c_1 \texttt{ else } c_2, Q)$ gelte.
  Zu zeigen: $\vdash \{\, \mathcal{I}\,[\![\mathrm{pre}(\texttt{if (}b\texttt{) then } c_1 \texttt{ else } c_2, Q)]\!] \,\}\, \mathrm{strip}(\texttt{if (}b\texttt{) then } c_1 \texttt{ else } c_2)\, \{\, \mathcal{I}\,[\![Q]\!] \,\}$.
  Nach Regel $\textsc{If}_\mathrm{P}$ und Definition von strip genügt es, folgende zwei Ableitungen zu zeigen:

  $$\{\, \lambda\sigma.\ \mathcal{B}\,[\![b]\!]\,\sigma \wedge \mathcal{I}\,[\![\mathrm{pre}(\texttt{if (}b\texttt{) then } c_1 \texttt{ else } c_2, Q)]\!]\,\sigma \,\}\, \mathrm{strip}(c_1)\, \{\, \mathcal{I}\,[\![Q]\!] \,\}$$
  $$\{\, \lambda\sigma.\ \neg\, \mathcal{B}\,[\![b]\!]\,\sigma \wedge \mathcal{I}\,[\![\mathrm{pre}(\texttt{if (}b\texttt{) then } c_1 \texttt{ else } c_2, Q)]\!]\,\sigma \,\}\, \mathrm{strip}(c_2)\, \{\, \mathcal{I}\,[\![Q]\!] \,\}$$

  Wegen $\mathrm{vc}(\texttt{if (}b\texttt{) then } c_1 \texttt{ else } c_2, Q) = \mathrm{vc}(c_1, Q)$ `&&` $\mathrm{vc}(c_2, Q)$ gelten auch $\mathrm{vc}(c_1, Q)$ und $\mathrm{vc}(c_2, Q)$, womit die Induktionsannahmen (für Q) anwendbar sind. Außerdem gilt für die Vorbedingungen:

  $$\begin{aligned}
  &\mathcal{B}\,[\![b]\!]\,\sigma \wedge \mathcal{I}\,[\![\mathrm{pre}(\texttt{if (}b\texttt{) then } c_1 \texttt{ else } c_2, Q)]\!]\,\sigma \\
  &\quad = \mathcal{B}\,[\![b]\!]\,\sigma \wedge \mathcal{I}\,[\![\texttt{(}b\texttt{ -> } \mathrm{pre}(c_1, Q)\texttt{) \&\& (not } b\texttt{ -> } \mathrm{pre}(c_2, Q)\texttt{)}]\!]\,\sigma \\
  &\quad = \mathcal{B}\,[\![b]\!]\,\sigma \wedge (\mathcal{B}\,[\![b]\!]\,\sigma \implies \mathcal{I}\,[\![\mathrm{pre}(c_1, Q)]\!]\,\sigma) \wedge (\neg\, \mathcal{B}\,[\![b]\!]\,\sigma \implies \mathcal{I}\,[\![\mathrm{pre}(c_2, Q)]\!]\,\sigma) \\
  &\quad = \mathcal{B}\,[\![b]\!]\,\sigma \wedge \mathcal{I}\,[\![\mathrm{pre}(c_1, Q)]\!]\,\sigma \implies \mathcal{I}\,[\![\mathrm{pre}(c_1, Q)]\!]\,\sigma
  \end{aligned}$$

  und entsprechend

  $$\neg\, \mathcal{B}\,[\![b]\!]\,\sigma \wedge \mathcal{I}\,[\![\mathrm{pre}(\texttt{if (}b\texttt{) then } c_1 \texttt{ else } c_2, Q)]\!] = \neg\, \mathcal{B}\,[\![b]\!]\,\sigma \wedge \mathcal{I}\,[\![\mathrm{pre}(c_2, Q)]\!]\,\sigma \implies \mathcal{I}\,[\![\mathrm{pre}(c_2, Q)]\!]\,\sigma$$

  Damit folgen die zu zeigenden Zusicherungen aus den Induktionsannahmen jeweils mit der Regel $\textsc{Cons}_\mathrm{P}$.

- Fall `while (`b`) {`I`} do` c: Induktionsannahme (für beliebiges Q):
  Wenn $\mathrm{vc}(c, Q)$, dann $\vdash \{\, \mathcal{I}\,[\![\mathrm{pre}(c, Q)]\!] \,\}\, \mathrm{strip}(c)\, \{\, \mathcal{I}\,[\![Q]\!] \,\}$.

  Sei Q beliebig, so dass $\mathrm{vc}(\texttt{while (}b\texttt{) \{}I\texttt{\} do } c, Q)$ gelte.
  Zu zeigen: $\vdash \{\, \mathcal{I}\,[\![\mathrm{pre}(\texttt{while (}b\texttt{) \{}I\texttt{\} do } c, Q)]\!] \,\}\, \mathrm{strip}(\texttt{while (}b\texttt{) \{}I\texttt{\} do } c)\, \{\, \mathcal{I}\,[\![Q]\!] \,\}$,
  also $\vdash \{\, \mathcal{I}\,[\![I]\!] \,\}\, \texttt{while (}b\texttt{) do } \mathrm{strip}(c)\, \{\, \mathcal{I}\,[\![Q]\!] \,\}$.

  Da $\mathrm{vc}(\texttt{while (}b\texttt{) \{}I\texttt{\} do } c, Q)$ gilt, gelten auch

(1) $\mathrm{vc}(c, I)$ sowie

(2) $\mathcal{B}[\![b]\!]\,\sigma \wedge \mathcal{I}[\![I]\!]\,\sigma \implies \mathcal{I}[\![\mathrm{pre}(c, I)]\!]\,\sigma$ und

(3) $\neg\,\mathcal{B}[\![b]\!]\,\sigma \wedge \mathcal{I}[\![I]\!]\,\sigma \implies \mathcal{I}[\![Q]\!]\,\sigma$ für alle σ.

Aus (1) folgt per Induktionsannahme $\vdash \{\,\mathcal{I}[\![\mathrm{pre}(c, I)]\!]\,\}\,\mathrm{strip}(c)\,\{\,\mathcal{I}[\![I]\!]\,\}$. Dies lässt wie folgt zusammenfügen:

$$\dfrac{\dfrac{\dfrac{(2)\qquad \vdash \{\,\mathcal{I}[\![\mathrm{pre}(c, I)]\!]\,\}\,\mathrm{strip}(c)\,\{\,\mathcal{I}[\![I]\!]\,\}}{\vdash \{\,\mathcal{B}[\![b]\!] \wedge \mathcal{I}[\![I]\!]\,\}\,\mathrm{strip}(c)\,\{\,\mathcal{I}[\![I]\!]\,\}}\ \textsc{Cons}_\textsc{p}}{\vdash \{\,\mathcal{I}[\![I]\!]\,\}\,\texttt{while (}b\texttt{) do strip(}c\texttt{)}\,\{\,\neg\,\mathcal{B}[\![b]\!] \wedge \mathcal{I}[\![I]\!]\,\}}\ \textsc{While}_\textsc{p} \qquad (3)}{\vdash \{\,\mathcal{I}[\![I]\!]\,\}\,\texttt{while (}b\texttt{) do strip(}c\texttt{)}\,\{\,\mathcal{I}[\![Q]\!]\,\}}\ \textsc{Cons}_\textsc{p}$$

□

Korollar 150. Wenn $\mathrm{vc}(c, Q)$ und P `->` $\mathrm{pre}(c, Q)$ gelten, dann gilt $\vdash \{\,P\,\}\,\mathrm{strip}(c)\,\{\,Q\,\}$.

Beispiel 151 (Fortsetzung von Bsp. 148). Sei $P =$ `x == 0 && i == 0`. Dann gelten P `->` I und $\mathrm{vc}(w, Q)$:

$$\begin{aligned}
\mathcal{B}[\![P \texttt{ -> } I]\!]\,\sigma &= (\texttt{x} = 0 \wedge \texttt{i} = 0 \implies 2 \cdot \texttt{x} = \texttt{i} \cdot (\texttt{i} + 1)) = \mathbf{tt} \\
\mathcal{B}[\![\mathrm{vc}(w, Q)]\!]\,\sigma &= (\texttt{i} \neq \texttt{n} \wedge 2 \cdot \texttt{x} = \texttt{i} \cdot (\texttt{i} + 1) \implies 2 \cdot (\texttt{x} + \texttt{i} + 1) = (\texttt{i} + 1) \cdot (\texttt{i} + 1 + 1)) \wedge \\
&\qquad (\texttt{i} = \texttt{n} \wedge 2 \cdot \texttt{x} = \texttt{i} \cdot (\texttt{i} + 1) \implies 2 \cdot \texttt{x} = \texttt{n} \cdot (\texttt{n} + 1)) \wedge \mathbf{tt} \wedge \mathbf{tt} \\
&= \mathbf{tt}
\end{aligned}$$

Somit gilt auch $\vdash \{\,P\,\}\,w\,\{\,Q\,\}$.

Korrektheit eines Verifikationsbedingungsgenerators (Thm. 149) sagt noch nichts darüber aus, ob dieser auch verwendbar ist. Die Verifikationsbedingung `false` wäre auch für alle Programme korrekt, ebenso wie die berechnete Vorbedingung `false`, aber diese beiden wären für die Verifikation von Programmen unbrauchbar. Wichtig ist deswegen die Umkehrung: Vollständigkeit.

Lemma 152 (Monotonie von $\mathrm{pre}(c, \_)$ und $\mathrm{vc}(c, \_)$).
Wenn $\mathcal{I}[\![P]\!] \implies \mathcal{I}[\![Q]\!]$, dann auch $\mathcal{I}[\![\mathrm{pre}(c, P)]\!] \implies \mathcal{I}[\![\mathrm{pre}(c, Q)]\!]$ und $\mathcal{I}[\![\mathrm{vc}(c, P)]\!] \implies \mathcal{I}[\![\mathrm{vc}(c, Q)]\!]$.

Beweis. Induktion über c (P und Q beliebig) und Ausrechnen. Wesentlich ist, dass P in $\mathrm{pre}(c, P)$ und $\mathrm{vc}(c, P)$ nur rechts der Implikationen auftritt. □

Theorem 153. Wenn $\vdash \{\,\mathcal{I}[\![P]\!]\,\}\,c\,\{\,\mathcal{I}[\![Q]\!]\,\}$, so dass alle Prädikate im Ableitungsbaum als Formeln darstellbar sind, dann gibt es ein annotiertes Programm c' mit $c = \mathrm{strip}(c')$, sodass $\mathrm{vc}(c', Q)$ und P `->` $\mathrm{pre}(c', Q)$ gelten.

Beweis. Induktion über $\vdash \{\,\mathcal{I}[\![P]\!]\,\}\,c\,\{\,\mathcal{I}[\![Q]\!]\,\}$:

- Fälle $\textsc{Skip}_\textsc{p}$, $\textsc{Ass}_\textsc{p}$: Trivial bei Wahl $c' = c$.
- Fall $\textsc{Seq}_\textsc{p}$: Induktionsannahmen: $\vdash \{\,\mathcal{I}[\![P]\!]\,\}\,c_1\,\{\,\mathcal{I}[\![Q]\!]\,\}$ und $\vdash \{\,\mathcal{I}[\![Q]\!]\,\}\,c_2\,\{\,\mathcal{I}[\![R]\!]\,\}$ und es gibt c_1' und c_2' mit $c_1 = \mathrm{strip}(c_1')$ und $c_2 = \mathrm{strip}(c_2')$, sodass $\mathrm{vc}(c_1', Q)$, P `->` $\mathrm{pre}(c_1', Q)$, $\mathrm{vc}(c_2', R)$ und Q `->` $\mathrm{pre}(c_2', R)$ gelten.

  Wähle $c' = c_1'$`;` c_2'. Da Q `->` $\mathrm{pre}(c_2', R)$ gilt, folgt aus den Induktionsannahmen mit der Monotonie (Lem. 152), dass auch $\mathrm{vc}(c', R) = \mathrm{vc}(c_1', \mathrm{pre}(c_2', R))$ und P `->` $\mathrm{pre}(c', R) = P$ `->` $\mathrm{pre}(c_1', \mathrm{pre}(c_2', R))$ gelten.
- Fall $\textsc{If}_\textsc{p}$: Induktionsannahmen: $\vdash \{\,\mathcal{I}[\![b$ `&&` $P]\!]\,\}\,c_1\,\{\,\mathcal{I}[\![Q]\!]\,\}$ und $\vdash \{\,\mathcal{I}[\![$`not` b `&&` $P]\!]\,\}\,c_2\,\{\,\mathcal{I}[\![Q]\!]\,\}$ und es gibt c_1' und c_2' mit $c_1 = \mathrm{strip}(c_1')$ und $c_2 = \mathrm{strip}(c_2')$, sodass $\mathrm{vc}(c_1', Q)$, b `&&` P `->` $\mathrm{pre}(c_1', Q)$, $\mathrm{vc}(c_2', Q)$ und `not` b `&&` P `->` $\mathrm{pre}(c_2', Q)$ gelten.

Wähle $c' = \texttt{if (}b\texttt{) then } c'_1 \texttt{ else } c'_2$. Dann gelten offensichtlich

$$\mathrm{vc}(c', Q) = \mathrm{vc}(c'_1, Q) \texttt{ \&\& } \mathrm{vc}(c'_2, Q)$$

und

$$P \texttt{ -> } \mathrm{pre}(c', Q) = P \texttt{ -> ((}b \texttt{ -> } \mathrm{pre}(c'_1, Q)\texttt{) \&\& (not } b \texttt{ -> } \mathrm{pre}(c'_2, Q)\texttt{))}$$

- Fall WhileP: Induktionsannahmen: $\vdash \{\, \mathcal{I}\,[\![b \texttt{ \&\& } I]\!] \,\}\, c \,\{\, \mathcal{I}\,[\![I]\!] \,\}$ und es gibt ein c^* mit $c = \mathrm{strip}(c^*)$, sodass $\mathrm{vc}(c^*, I)$ und $b \texttt{ \&\& } I \texttt{ -> } \mathrm{pre}(c^*, I)$ gelten.

  Wähle $c' = \texttt{while (}b\texttt{) \{}I\texttt{\} do } c^*$. Dann gelten offensichtlich

  $$\mathrm{vc}(c', \texttt{not } b \texttt{ \&\& } I) = \texttt{(}b \texttt{ \&\& } I \texttt{ -> } \mathrm{pre}(c^*, I)\texttt{) \&\& (not } b \texttt{ \&\& } I \texttt{ -> not } b \texttt{ \&\& } I\texttt{) \&\& } \mathrm{vc}(c^*, I)$$

  und

  $$I \texttt{ -> } \mathrm{pre}(c', \texttt{not } b \texttt{ \&\& } I) = I \texttt{ -> } I$$

- Fall ConsP: Induktionsannahme: $\vdash \{\, \mathcal{I}\,[\![P']\!] \,\}\, c \,\{\, \mathcal{I}\,[\![Q']\!] \,\}$ und es gibt ein c' mit $c = \mathrm{strip}(c')$, sodass $\mathrm{vc}(c', Q')$ und $P' \texttt{ -> } \mathrm{pre}(c', Q')$ gelten. Außerdem $\mathcal{I}\,[\![P]\!] \implies \mathcal{I}\,[\![P']\!]$ und $\mathcal{I}\,[\![Q']\!] \implies \mathcal{I}\,[\![Q]\!]$.

  Wegen der Monotonie (Lem. 152) gelten auch $\mathrm{vc}(c', Q)$ und $P' \texttt{ -> } \mathrm{pre}(c', Q)$. Da $P \texttt{ -> } P'$, gilt auch $P \texttt{ -> } \mathrm{pre}(c', Q)$. □

In Thm. 153 ist ganz wesentlich, dass alle Prädikate des Ableitungsbaums in der Formelsprache ausdrückbar sind. Ansonsten gibt es Fälle (Bsp. 144), bei denen zwar Vor- und Nachbedingung als Formel ausgedrückt werden können, aber nicht die benötigten Zwischenbedingungen. Wären von Anfang an nur Formeln als Vor- und Nachbedingung zugelassen, gäbe es dieses Problem nicht. Bei praktischen Anwendungen von Verifikationsbedingungsgeneratoren definiert man üblicherweise das Hoare-Kalkül so, dass Vor- und Nachbedingungen syntaktische Formeln sind.

Korollar 154. Sei die Formelsprache expressiv. Wenn $\vdash \{\, \mathcal{I}\,[\![P]\!] \,\}\, c \,\{\, \mathcal{I}\,[\![Q]\!] \,\}$, dann gibt es ein c' mit $c = \mathrm{strip}(c')$, sodass $\mathrm{vc}(c', Q)$ und $P \texttt{ -> } \mathrm{pre}(c', Q)$ gelten.

Beweis. Aus $\vdash \{\, \mathcal{I}\,[\![P]\!] \,\}\, c \,\{\, \mathcal{I}\,[\![Q]\!] \,\}$ gilt nach Thm. 137 (Korrektheit), dass $\vDash \{\, \mathcal{I}\,[\![P]\!] \,\}\, c \,\{\, \mathcal{I}\,[\![Q]\!] \,\}$. Nach dem Beweis der Vollständigkeit (Thm. 142) gibt es eine Ableitung $\vdash \{\, \mathcal{I}\,[\![P]\!] \,\}\, c \,\{\, \mathcal{I}\,[\![Q]\!] \,\}$, in der nur schwächste freie Vorbedingungen vorkommen, für die es nach Annahme Formeln gibt. Damit ist Thm. 153 anwendbar.

Der Umweg über Korrektheit und Vollständigkeit ist notwendig, weil der Ableitungsbaum von $\vdash \{\, \mathcal{I}\,[\![P]\!] \,\}\, c \,\{\, \mathcal{I}\,[\![Q]\!] \,\}$ Prädikate enthalten kann, die nicht als Formel darstellbar sind. Expressivität fordert schließlich nur, dass die schwächsten freien Vorbedingungen ausdrückbar sind. □

9 Exkurs: Semantik funktionaler Programmiersprachen

Bisher haben wir uns eine typische imperative Programmiersprache angeschaut: Globaler veränderlicher Zustand und explizite Kontrollstrukturen. In diesem Kapitel machen wir einen kleinen Ausflug in die Welt der funktionalen Programmierung. Der Ausflug ist ein relativ seichter Einstieg und behandelt das Thema weder in der Breite, noch in der Tiefe.

9.1 Das Lambda-Kalkül

Auch hier werden wir uns wieder ein Kern-Kalkül betrachten, das nur die wichtigen Eigenschaften einer funktionalen Programmiersprache enthält, und sonst nichts. Ein solches Kern-Kalkül ist das Lambda-Kalkül:

9.1.1 Syntax

Definition 155. Syntax des Lambda-Kalküls Die Syntax von Ausdrücken des Lambda-Kalküls sei durch folgende Grammatik gegeben:

$$\mathsf{\lambda Exp} \quad e \quad ::= \quad x \mid e\, e \mid \lambda x.\, e$$

Ein Lambda-Ausdruck ist also entweder eine Variable, oder eine Funktionsanwendung, oder eine Lambda-Abstraktion.

Wie bisher werden wir Klammern setzen, wenn die Syntax sonst uneindeutig wird. Funktionsanwendung assoziiert nach links: $e_1\, e_2\, e_3 = (e_1\, e_2)\, e_3$.

Die Variable x in $\lambda x.\, e$ ist *gebunden* mit Gültigkeitsbereich e. Wir werden uns hier nicht mit der Behandlung von Namensbindung aufhalten und sehen α-äquivalente Terme, also solche, die durch Umbenennung von gebundenen Variablen auseinander vorgehen, als gleich an: $\lambda x.\, x = \lambda y.\, y$.

Eine Auftreten einer Variable x in einem Term ist *frei*, wenn es nicht innerhalb einer Lambda-Abstraktion $\lambda x.\, e$ steht.

Definition 156 (Substitution). Der Ausdruck $e_1[e_2/x]$ bezeichnet die *Substitution* (Ersetzung) von jedem freien x durch e_2.

Beispiel 157. Es ist

$$\begin{aligned}((\lambda y.\, (x\, (\lambda x.\, (x\, y))))\, x)[y/x] &= ((\lambda y.\, (x\, (\lambda x.\, (x\, y))))[y/x])\, y \\ &= (\lambda y'.\, ((x\, (\lambda x.\, (x\, y')))[y/x]))\, y \\ &= (\lambda y'.\, (y\, ((\lambda x.\, (x\, y'))[y/x])))\, y \\ &= (\lambda y'.\, (y\, (\lambda x.\, (x\, y'))))\, y.\end{aligned}$$

Man beachte dass die gebundene Variable y zu y' umbenannt wurde, damit sie nicht mit der Variable y, die substituiert wird, verwechselt wird. Auch mit diesem Phänomen werden wir uns nicht weiter aufhalten.

9.1.2 β-Reduktion

Die Intuition hinter den jeweiligen Arten von Lambda-Ausdrücken ist klar: Eine Lambda-Abstraktion definiert eine Funktion, während eine Funktionsanwendung eine solche auf ein Argument anwendet. Wird eine Lambda-Abstraktion direkt angewendet, so nennt man dies β-Reduktion. Mit diesem als elementaren Schritt definieren wir eine Small-Step-Semantik für das Lambda-Kalkül:

Definition 158 (β-Reduktion). Die Ein-Schritt-Reduktionsrelation $\to_\beta$ ist induktiv definiert durch die β-Reduktion und Kongruenzregeln:

$$\textsc{Beta}\colon (\lambda x.\, e_1)\, e_2 \to_\beta e_1[e_2/x]$$

$$\textsc{Cong-App-L}\colon \frac{e_1 \to_\beta e_2}{e_1\, e \to_\beta e_2\, e} \qquad \textsc{Cong-App-R}\colon \frac{e_1 \to_\beta e_2}{e\, e_1 \to_\beta e\, e_2} \qquad \textsc{Cong-Lam}\colon \frac{e_1 \to_\beta e_2}{\lambda x.\, e_1 \to_\beta \lambda x.\, e_2}$$

Es ist $\to_\beta^*$ die reflexive, transitive Hülle von $\to_\beta$, und $=_\beta$ die dadurch erzeugte Äquivalenzrelation.

Definition 159 (Normalform). Ein Lambda-Ausdruck e ist in *Normalform*, wenn es kein e' gibt, so dass $e \to_\beta e'$. Äquivalent: Kein Teilterm von e ist ein Redex $(\lambda x.\, e_1)\, e_2$.

Ein Lambda-Ausdruck e *hat* eine Normalform e' wenn $e =_\beta e'$ und e' in Normalform ist. In diesem Fall heißt e (schwach) normalisierbar.

Eine erste wichtige Fragestellung ist nun: Sind alle Lambda-Terme normalisierbar? Dem ist nicht so:

Beispiel 160 (Oh! Mega-Kombinator). Der Lambda-Ausdruck $\Omega \coloneqq (\lambda x.\, x\, x)\, (\lambda x.\, x\, x)$ hat keine Normalform, denn $\Omega \to_\beta \Omega \to_\beta \Omega \cdots$.

Ein Termersetzungssystem als ganzes heißt normalisierbar, wenn alle Terme normalisierbar sind. Das Lambda-Kalkül ist es also nicht. Würden wir allerdings nur einfach typisierte Lambda-Ausdrücke betrachten, so wäre das Kalkül normalisierbar.

Ist wenigstens für einen normaliserbaren Terme die Normalform eindeutig? Das ist in der Tat der Fall:

Lemma 161 (Eindeutigkeit der Normalform). Ist sowohl e_1 als auch e_2 eine Normalform von e, so gilt $e_1 = e_2$.

Beweis. Siehe Vorlesung Programmierparadigmen. Ein wichtiger Beweisschritt dabei ist die Feststellung, dass $\to_\beta^*$ konfluent ist: Ist $e \to_\beta^* e_1$ und $e \to_\beta^* e_2$, so gibt es e_3 mit $e_1 \to_\beta^* e_3$ und $e_2 \to_\beta^* e_3$. Dies nennt man auch die Church-Rosser-Eigenschaft. □

Die Relevanz dieses Lemma ergibt sich aus folgenden einfachen Korrolar:

Korollar 162 (Konsistenz des Lambda-Kalküls). Die von der Äquivalenzrelation $=_\beta$ definierte Theorie ist Konsistent: Es gibt Terme e_1 und e_2 mit $e_1 \neq_\beta e_2$.

Beweis. Für $e_1 =_\beta e_2$ gilt, per Induktion über die Regeln der reflexiven, symmetrischen und transitiven Hülle, dass es ein e' gibt mit $e_1 \to_\beta^* e'$ und $e_2 \to_\beta^* e'$.

Wähle verschiedene Normalformen e_1 und e_2 (z.B. $\lambda x.\, \lambda y.\, x$ und $\lambda x.\, \lambda y.\, y$). Angenommen, $e_1 =_\beta e_2$. Dann gibt es ein e' mit $e_1 \to_\beta^* e'$ und $e_2 \to_\beta^* e'$, also $e_1 = e' = e_2$, was nicht stimmt. Damit ist $e_1 \neq_\beta e_2$. gezeigt. □

9.2 Ein Modell für das Lambda-Kalkül

Im vorigen Abschnitt haben wir eine Semantik für Lambda-Ausdrücke gefunden: Die Relation $=_\beta$ gibt an, wann zwei Lambda-Ausdrücke gleich sind.

Allerdings ist das eine sehr operationelle Sicht, was nicht befriedigend ist. Wir möchten also gerne die Bedeutung eines Lambda-Ausdrucks über ein eigenständiges mathematisches Objekt definieren.

Lambda-Ausdrücke sind Funktionen, und so ist es naheliegend, als Modell eine Menge X zu wählen,die groß genug ist, alle Funktionen $X \to X$ (etwa über eine Einbettung) zu enthalten. Leider gibt es keine solche Menge, die nicht trivial ist: Sofern X mindestens zwei Elemente enthält, hat $X \to X$ eine Kardinalität, die mindestens so groß ist wie die von $\mathcal{P}(X)$, der Potenzmenge von X, welche eine echt größere Kardinalität als X hat. Es gibt also zu viele Funktionen $X \to X$, als dass wir sie alle in X finden können.

Ein Schritt zur Lösung ist die Erkenntnis, dass wir gar nicht alle Funktionen $X \to X$ brauchen! Schließlich sind Lambda-Ausdrücke Programme, und können nur berechenbare Funktionen darstellen, und es sind bei weitem nicht alle Funktionen berechenbar.

Wir greifen dazu auf die Domänentheorie zurück, die wir bereits in Kapitel 7 kennen gelernt haben. Konkret suchen wir eine ccpo D mit einer Funktion

$$\varphi\colon (D \to D) \to D,$$

mit der sich stetige Funktionen in D einbetten lassen, sowie einer Funktion

$$\psi\colon D \to D \to D,$$

mit der sich ein Element aus D als Funktion interpretieren lässt. Es sollte $\psi \circ \varphi$ die Identität auf $D \to D$ sein.

Man hätte statt Domäne auch allgemeiner Kategorie sagen können; kartesisch abgeschlossene Kategorien mit solchen Morphismen φ und ψ heißen *reflexiv* und können alle als Modell für das Lambda-Kalkül dienen.

Derer Domänen gibt es viele; wir beschränken uns auf die kleinste, für die gilt

$$D \equiv [D \to D]_\bot.$$

Dabei ist $A_\bot = A \cup \{\bot\}$ die Halbordnung, die aus A hervorgeht, wenn man es um ein dediziertes kleines Element erweitert. Bei Bedarf verwenden wir ausdrücklich die Einbettung $\mathrm{up}\colon A \to A_\bot$

Lemma 163 (Konstruktion von D). Es gibt eine ccpo D mit $D \equiv [D \to D]_\bot$.

Beweis. Wir folgen Abramsky (1993) und bauen eine Folge von Approximationen von D auf. Es sei also $D_0 = \{\bot\}$ und $D_{n+1} \coloneqq [D_n \to D_n]_\bot$. Wir können D_n in D_{n+1} einbetten mittels

$$i_n\colon D_n \to D_{n+1} \quad \text{und} \quad j_n\colon D_{n+1} \to D_n$$

wobei

$$\begin{aligned} i_0(\bot_{D_0}) &= \bot_{D_1} \\ j_0(d) &= \bot_{D_0} \\ i_{n+1}(d) &= \begin{cases} \bot_{D_{n+2}} & \text{falls } d = \bot_{D_{n+1}} \\ \mathrm{up}(i_n \circ f \circ j_n) & \text{falls } d = \mathrm{up}(f) \text{ mit } f \in D_n \to D_n \end{cases} \\ j_{n+1}(d) &= \begin{cases} \bot_{D_{n+1}} & \text{falls } d = \bot_{D_{n+2}} \\ \mathrm{up}(j_n \circ f \circ i_n) & \text{falls } d = \mathrm{up}(f) \text{ mit } f \in D_{n+1} \to D_{n+1} \end{cases} \end{aligned}$$

Nun konstruieren wir

$$D \coloneqq \big\{ d \in \prod_{n\in\mathbb{N}} D_n \mid j_n(d_{n+1}) = d_n \big\}.$$

Für $d \in D$ gilt $d_n = \bot$ entweder für alle oder für kein $n > 0$.

Wir können ein $d \in D_n$ auffassen als Element von D, indem wir die Einträge für $m < n$ mittels der Funktionen j_m, $m < n$, und die Einträge für $m > n$ mittels der Funktionen i_m, $m \geq n$ basteln, und bezeichnen dies mit $i_n^\infty \colon D_n \to D$. Es gilt $d = \bigsqcup \{ i_n^\infty(d_n) \mid n \in \mathbb{N} \}$ für alle $d \in D$.

Erfüllt dies die gewünschte Gleichung? Wir haben zum einen $i : D \to [D \to D]_\bot$ mit

$$i(d) = \begin{cases} \bot & \text{falls } d = \bot \\ \mathrm{up}\left(\lambda d'.(f_{n+1}(d'_n))_{n\in\mathbb{N}}\right) & \text{falls } d_n = \mathrm{up}(f_n) \text{ für } n > 0 \end{cases}$$

als auch $j : [D \to D]_\bot \to D$ mit

$$j(d) = \begin{cases} (\bot)_{n\in\mathbb{N}} & \text{falls } d = \bot \\ \big(\mathrm{up}(\lambda d' \in D_{n-1}.(f(i_{n-1}^\infty(d')))_{n-1})\big)_{n>0} & \text{falls } d = \mathrm{up}(f). \end{cases}$$

Diese Funktionen realisieren den gewünschten Isomorphismus. Rechnet man die Isomorphismen-Eigenschaft nach, wird man feststellen, dass man für $i \circ j = id$ die Stetigkeit der Funktionen $D \to D$ braucht:

Sowohl i als auch j sind strikt, es genügt also $i(j(\mathrm{up}\, f)) = \mathrm{up}\, f$ für eine stetige Funktion $f \colon D \to D$ nachzurechnen. Es ist $i(j(\mathrm{up}\, f)) = \mathrm{up}(\lambda d'.(f_{n+1}(d'_n))_{n\in\mathbb{N}})$, wobei $f_n(d') \coloneqq (f(i_{n-1}^\infty(d')))_{n-1} \colon D_n \to D_n$. Sei also $d' \in D$ und wir sehen

$$\begin{aligned} (\lambda d'.(f_{n+1}(d'_n))_{n\in\mathbb{N}})\, d' &= (f_{n+1}(d'_n))_{n\in\mathbb{N}} \\ &= \bigsqcup \{ i_n^\infty(f_{n+1}(d'_n)) \mid n \in \mathbb{N} \} \\ &= \bigsqcup \{ i_n^\infty((f(i_n^\infty(d'_n)))_n) \mid n \in \mathbb{N} \} \\ &= \bigsqcup \{ f(i_n^\infty(d'_n)) \mid n \in \mathbb{N} \} \qquad \{ \text{á la Diagonallemma } \} \\ &= f(\bigsqcup \{ i_n^\infty(d'_n) \mid n \in \mathbb{N} \}) \\ &= f(d') \end{aligned}$$

Die oben gesuchten Funktionen $\varphi \colon (D \to D) \to D$ und $\psi \colon D \to D \to D$ sind nun gegeben durch

$$\varphi(f) = j(\mathrm{up}(f)) \text{ und } \psi(d)(d') = \begin{cases} \bot & \text{falls } i(d) = \bot \\ f(d') & \text{falls } i(d) = \mathrm{up}(f). \end{cases}$$

□

Statt $\psi(d)(d')$ schreiben wir auch $d \cdot d'$.

Nun können wir endlich unseren Lambda-Termen eine Bedeutung in D zuweisen.

Definition 164 (Denotationale Semantik der Lambda-Ausdrücke). Die Semantik eines Lambda-Ausdrucks e in einer *Umgebung* $\rho\colon \mathsf{Var} \to D$ ist gegeben durch $[\![e]\!]_\rho \in D$

$$\begin{aligned}[\![x]\!]_\rho &\coloneqq \rho\, x \\ [\![e_1\, e_2]\!]_\rho &\coloneqq [\![e_1]\!]_\rho \cdot [\![e_2]\!]_\rho \\ [\![\lambda x.\, e]\!]_\rho &\coloneqq \varphi(\lambda d.\, [\![e]\!]_{\rho[x \mapsto d]})\end{aligned}$$

Lemma 165 (Substitutionslemma). Es gilt $[\![e_1]\!]_{\rho[x \mapsto [\![e_2]\!]_\rho]} = [\![e_1[e_2/x]]\!]_\rho$.

Lemma 166. Die Beta-Reduktion $\to_\beta$ ist korrekt bezüglich der Semantik: Für $e_1 \to_\beta e_2$ ist $[\![e_1]\!]_\rho = [\![e_2]\!]_\rho$ für alle Umgebungen ρ.

Beweis. Per Induktion über die Regeln von $\to_\beta$, mit beliebigen ρ.

- Regel Beta: Es gilt

$$\begin{aligned}[\![(\lambda x.\, e_1)\, e_2]\!]_\rho &= [\![\lambda x.\, e_1]\!]_\rho \cdot [\![e_2]\!].\, \rho \\ &= \varphi(\lambda d.\, [\![e_1]\!]_{\rho[x \mapsto d]}) \cdot [\![e_2]\!].\, \rho \\ &= [\![e_1]\!]_{\rho[x \mapsto [\![e_2]\!]_\rho]} \\ &= [\![e_1[e_2/x]]\!]_\rho && \{ \text{ Lemma 165 } \}\end{aligned}$$

- Regeln Cong-App-L, Cong-App-R und Cong-Lam: Es gilt

$$[\![e_1\, e]\!]_\rho = [\![e_1]\!]_\rho \cdot [\![e]\!]_\rho = [\![e_2]\!]_\rho \cdot [\![e]\!]_\rho = [\![e_2\, e]\!]_\rho \tag{1}$$

$$[\![e\, e_1]\!]_\rho = [\![e]\!]_\rho \cdot [\![e_1]\!]_\rho = [\![e]\!]_\rho \cdot [\![e_2]\!]_\rho = [\![e\, e_2]\!]_\rho \tag{2}$$

$$[\![\lambda x.\, e_1]\!]_\rho = \varphi(\lambda d.\, [\![e_1]\!]_{\rho[x \mapsto d]}) = \varphi(\lambda d.\, [\![e_2]\!]_{\rho[x \mapsto d]}) = [\![\lambda x.\, e_2]\!]_\rho \tag{3}$$

  wobei die mittlere Gleichung jeweils die Induktionsannahme verwendet. □