Semantik und Verifikation
WS 97/98

Martin Hofmann
FB 4, TU-Darmstadt
Schlossgartenstraße 7
64289 Darmstadt
mh@mathematik.tu-darmstadt.de

1 Einführung

Jede(r), der/die schon einmal programmiert hat, macht folgende zwei Erfahrungen.

- Programme enthalten Fehler
- Das Finden und Beheben von Fehlern beansprucht meistens mehr Zeit, als das eigentliche Programmieren.

Manche Fehler werden trotz umfangreicher Suche nicht gefunden und verursachen dann Folgen, die von einfachen Funktionsstörungen („Tut mir leid, der Computer streikt, kommen Sie bitte morgen wieder") bis hin zu Katastrophen (Ariane-Crash, Raketenunfall im Golfkrieg) reichen. Ein wichtiges Ziel von Informatikforschung ist es daher, Methoden bereitzustellen, die helfen, Programmierfehler zu vermeiden. Im Prinzip gibt es dazu zwei Möglichkeiten:

- Bessere Programmiersprachen
  - Strukturierung durch abstrakte Konzepte (Blockstruktur statt „Goto", objektorientierte Programmierung, deklarative Programmierung, Modularisierung)
  - Typisierung
  - Verwendung von Bibliotheken und Standardsoftware
- Verifikation
  - Dokumentation
  - Debugger

- Erstellung von Spezifikationen
- Beweis der Programmkorrektheit (von informell über formal bis rechnergestützt).

Die Semantik von Programmiersprachen, also das Thema dieser Vorlesung, ist bei beiden Ansätzen von Bedeutung. Ziel der Semantik ist eine mathematische Beschreibung der Bedeutung von Programmen und Programmstücken. Hat man eine Semantik festgelegt, so wird die Aufgabe, ein Programm zu spezifizieren und zu verifizieren zu einem wohldefinierten Problem. (Natürlich muss man dann wieder informell argumentieren, dass Semantik und Spezifikation das sind, was man möchte ...). Semantik hilft aber auch, programmiersprachliche Konstrukte besser zu verstehen und daher fehleranfällige Konstrukte durch klarere zu ersetzen (bessere Programmiersprachen).

Drittens dient Semantik zur maschinenunabhängigen Spezifikation der Bedeutung einer Programmiersprache und erlaubt damit Korrektheitsbeweise für Compiler. Es ist sogar möglich, aus einer formalen Semantik einen Compiler automatisch zu generieren.

Traditionell unterscheidet man zwischen *operationaler Semantik*, *denotationaler Semantik* und *axiomatischer Semantik*.

Unter einer operationalen Semantik versteht man die Beschreibung einer Programmiersprache durch Auswerteregeln oder eine abstrakte Maschine. Im weitesten Sinne kann man auch einen Compiler als operationale Semantik auffassen. Meistens ist aber die operationale Semantik abstrakter gefasst und kann bei der Implementierung eines Interpreters oder Compilers als Richtschnur dienen.

Denotationale Semantik weist einem Programm eine Funktion zu, welche die Wirkung des Programmes auf Eingaben beschreibt. Diese Funktion wird durch Induktion über den Programmaufbau definiert. Die Bedeutung der denotationalen Semantik liegt darin, dass sie für Programmverifikation mit mathematischen Methoden besser geeignet ist. Damit das Sinn macht, muss man natürlich die denotationale Semantik mit der operationalen in Bezug setzen. Wie das genau geht, ist eines unserer Lernziele. Hauptanwendungsgebiet denotationaler Semantik liegt bei funktionalen (oder deklarativen) Programmiersprachen.

Axiomatische Semantik ist noch näher an der Verifikation. Unter ihr versteht man einen Satz von Beweisregeln für die Korrektheit von Programmen in einer bestimmten Programmiersprache. Man kann darüber streiten, inwiefern ein solches Beweissystem als „Semantik“ zu bezeichnen ist. In der Vorlesung werden wir eine axiomatische Semantik kennenlernen, welche vollständig ist in dem Sinne, dass alle wahren Korrektheitsaussagen auch herleitbar sind. Das Beweissystem beschreibt daher die Bedeutung von Programmen so genau wie irgend möglich und verdient deshalb die Bezeichnung „Semantik“. Die axiomatische Semantik spielt vor allem im Zusammenhang mit imperativen Programmiersprachen eine Rolle.

1.1 Überblick

In der Vorlesung werden wir alle drei Arten von Semantikgebung kennenlernen und teilweise miteinander in Verbindung setzen. Der Schwerpunkt liegt auf den Anwendungen für Verifikationszwecke. Die Vorlesung gliedert sich in drei Teile.

Wir beginnen mit einer einfachen Pascal-artigen Programmiersprache für die wir eine operationale Semantik mit Hilfe von Zustandsübergängen definieren werden. Die Semantik boole'scher und arithmetischer Ausdrücke, die Teil der Sprache sind, werden wir dagegen denotational fassen. Wir werden dann einen Verifikationskalkül (axiomatische Semantik) angeben und dessen Korrektheit und Vollständigkeit bezüglich der operationalen Semantik beweisen. Die imperative Programmiersprache werden wir dann um rekursive Prozeduren erweitern und operationale Semantik wie Verifikationskalkül entsprechend anpassen.

Unser nächstes Ziel wird die Entwicklung von operationaler und denotationaler Semantik für eine applikative Programmiersprache mit Funktionen höherer Ordnung und Rekursion sein. Die operationale Semantik geben wir hier durch Auswerteregeln, die beschreiben, wie ein Programmstück zu einem Wert (eine Zahl oder ein Funktionsausdruck) vereinfacht werden kann. Die denotationale Semantik basiert auf sogenannten *domains* (bestimmte partielle Ordnungen). Als Vorbereitung werden wir daher einige Grundlagen der Domaintheorie behandeln.

Obwohl es für Programmiersprachen mit Funktionen höherer Ordnung keine vollständige axiomatische Semantik zu geben scheint, kann man aus der denotationalen Semantik verschiedene Beweisregeln herleiten, die wir anhand von Beispielen einsetzen werden.

Das entscheidende Resultat in diesem Teil ist die *Adäquatheit* der denotationalen Semantik; sie besagt, dass denotational gleiche Programmstücke in jedem Programmkontext gleiche Resultate unter der operationalen Semantik liefern. Einerseits liefert das eine gewisse Rechtfertigung der operationalen Semantik, andererseits zeigt es, dass aus der denotationalen Semantik abgeleitete Beweisregeln korrekt sind.

Im dritten Teil der Vorlesung (falls dann noch Zeit ist ...) werden wir ein oder mehrere Vertiefungsthemen behandeln. Möglich sind z.B. rekursiv definierte und abstrakte Datentypen, Semantik lokaler Variablen und Algol-Theorie, oder Semantik paralleler Prozesse und *model checking*.

1.2 Literatur

Unsere Hauptreferenz für die ersten zwei Vorlesungsteile ist

> Glynn Winskel. *The Formal Semantics of Programming Languages, an Introduction.* MIT Press, 1993.

Weitere Standardwerke sind

Carl Gunter. *Semantics of Programming Languages, Structures and Techniques.* MIT Press, 1992.

R. D. Tennent. *Semantics of Programming Languages.* Prentice Hall, 1991.

John Mitchell. *Foundations for Programming Languages.* MIT Press, 1996.

Hanne Riis Nielson und Flemming Nielson. *Semantics with Applications. A Formal Introduction.* Wiley, 1992.

Im Skripttext werde ich gelegentlich auf relevante Originalarbeiten hinweisen. Das Skript erhebt keinen Anspruch auf Originalität; bis auf einige Anpassungen und didaktische Kniffe entstammen die behandelten Resultate sämtlich der Literatur, auch wenn darauf nicht explizit hingewiesen wird.

2 Eine imperative Programmiersprache

Die Sprache IMP ist ein Fragment von Pascal, C, oder Algol mit *integer*-Variablen, arithmetischen und booleschen Ausdrücken und den Kontrollstrukturen Hintereinanderausführung, Wertzuweisung, Fallunterscheidung und *while*-Schleife.

Die Sprache IMP ist vollständig in dem Sinne, dass sich darin alle berechenbaren Funktionen programmieren lassen, aber natürlich praktisch etwas unbefriedigend. Der Vorteil von IMP ist, dass die Semantik überschaubar ist und der Blick auf das Wesentliche nicht verstellt wird. Manche fehlende Konstrukte, wie *repeat* und *for*-Schleife sind reine definitorische Erweiterungen (sog. *syntactic sugar*), andere, wie etwa reichere Datentypen und Ein-/Ausgabe lassen sich ohne konzeptuelle Neuarbeit aufnehmen; wieder andere, wie z.B. Prozeduren und lokale Variablen erfordern einen neuen Ansatz bei der Semantikgebung.

2.1 Syntax

Die Syntax der Sprache besteht aus folgenden syntaktischen Kategorien:

- numerische Konstanten **int**,
- Wahrheitswerte **bool**,
- Programmvariablen **var**,
- Arithmetische Ausdrücke **aexp**,
- Boolesche Ausdrücke **bexp**,
- Befehle oder Programme **com**

Numerische Konstanten bezeichnen positive und negative ganze Zahlen, sie können z.B. im Dezimalsystem angegeben werden. Wahrheitswerte sind **true** und **false**. Programmvariablen sind im Prinzip eine beliebige Menge; in den Beispielen nehmen wir Zeichenketten in `Schreibmaschine` beginnend mit einem Großbuchstaben. Wohlgeformte arithmetische und boolesche Ausdrücke, sowie Befehle definieren wir durch eine BNF Grammatik. Wir verwenden dabei folgende Metavariablen:

- m, n für numerische Konstanten **int**,
- X, Y für Programmvariablen **var**,
- a für arithmetische Ausdrücke **aexp**,
- b für boolesche Ausdrücke **bexp**,
- c für Befehle oder Programme **com**

Die Metavariablen dekorieren wir mit Strichen, Sub- und Superskripten. Jetzt also die Grammatik.

$$\begin{aligned}
a &::= n \mid \mathbf{var} \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \cdot a_2 \\
b &::= \mathbf{true} \mid \mathbf{false} \mid a_1{=}a_2 \mid a_1{\leq}a_2 \mid \neg b \mid b_1 \wedge b_2 \\
c &::= \mathbf{skip} \mid X{:=}a \mid c_1; c_2 \mid \mathbf{if}\, b\, \mathbf{then}\, c_1\, \mathbf{else}\, c_2 \mid \mathbf{while}\, b\, \mathbf{do}\, c
\end{aligned}$$

Diese Regeln spezifizieren die *abstrakte Syntax* von IMP und entsprechen der Ausgabe eines *parsers*. Für die konkrete Syntax, also die, die man tatsächlich eintippt, müßte man sich Gedanken machen über Vorrang- und Klammerregeln (Punkt vor Strich, etc.), sowie über die konkrete ASCII-Realisierung der Symbole. Wir verzichten auf die Erarbeitung so einer konkreten Syntax und verwenden in Beispielen die abstrakte Syntax, wobei wir Mehrdeutigkeiten durch Klammerungen, Einrückungen, oder durch den Zusammenhang beseitigen. Hier ist so ein Beispielprogramm.

$$\begin{array}{l}
\mathtt{A} := 0; \\
\mathtt{B} := \mathtt{X}; \\
\mathbf{if}\ \mathtt{Y} = 0\ \mathbf{then\ skip\ else} \\
\quad \mathbf{while}\ \mathtt{Y} \leq \mathtt{B}\ \mathbf{do} \\
\qquad \mathtt{B} := \mathtt{B} - \mathtt{Y}; \\
\qquad \mathtt{A} := \mathtt{A} + 1
\end{array}$$

Dieses Programm, bezeichnen wir es mit c_{div}, berechnet Quotienten und Rest der ganzzahligen Division $\mathtt{X}/\mathtt{Y}$, falls $\mathtt{X} > 0$ und $\mathtt{Y} > 0$. Man beachte, dass Ein/Ausgabe hier durch freie Programmvariablen realisiert wird. Man beachte auch, dass in IMP kein Unterschied zwischen Programmen und Programmstücken gemacht wird, da keine Deklarationen erforderlich sind.

2.2 Zustände

Zunächst wollen wir die Bedeutung boolescher und arithmetischer Ausdrücke festlegen. Da diese Programmvariablen enthalten können, hängt der Wert eines solchen Ausdrucks von der Belegung der Programmvariablen ab. Die Belegung aller Programmvariablen bezeichnet man als Zustand. Die Menge Σ der Zustände ist also die Menge $\mathbf{var} \to \mathbf{int}$ der Funktionen von $\mathbf{var}$ nach $\mathbf{int}$. Ist $\sigma \in \Sigma$ ein Zustand und $X \in \mathbf{var}$ eine Programmvariable, so bezeichnet $\sigma(X)$ den Wert dieser Variablen im Zustand σ. Ist weiter $n \in \mathbf{int}$, so schreiben wir $\sigma[X{\mapsto}n]$ für den durch

$$\sigma[X{\mapsto}n](Y) \stackrel{\text{def}}{=} \begin{cases} \sigma(Y), \text{falls } X \neq Y \\ n, \text{falls } X = Y \end{cases}$$

definierten Zustand.

Wir können **Init** als Bezeichnung für den durch $\mathbf{Init}(X) = 0$ definierten Zustand einführen und dann konkrete Zustände hinschreiben. Zum Beispiel ist

$$\mathbf{Init}[\mathtt{X}{\mapsto}15][\mathtt{Y}{\mapsto}4][\mathtt{A}{\mapsto}2][\mathtt{B} \mapsto 7]$$

ein Zustand, der in obigem Beispielprogramm nach zwei Durchläufen erreicht werden kann.

2.3 Denotationale Semantik von Ausdrücken

Die Semantik eines arithmetischen/booleschen Ausdrucks a bzw. b ist eine Funktion $[\![a]\!]$, bzw. $[\![b]\!]$ von Zuständen nach $\mathbf{int}/\mathbf{bool}$. Diese Funktionen definieren wir wie folgt durch Induktion über die Syntax von Ausdrücken.

$$\begin{array}{lcl}
[\![n]\!]\sigma & = & n \\
[\![X]\!]\sigma & = & \sigma(X) \\
[\![a_1{\star}a_2]\!]\sigma & = & [\![a_1]\!]\sigma \star [\![a_2]\!]\sigma, \quad \text{wobei } \star \in \{+,-,\cdot\} \\
 & & \\
[\![b]\!]\sigma & = & b \\
[\![a_1{=}a_2]\!]\sigma & = & \begin{cases} \mathbf{true}, \text{falls } [\![a_1]\!]\sigma = [\![a_2]\!]\sigma \\ \mathbf{false}, \text{sonst} \end{cases} \\
[\![a_1{\leq}a_2]\!]\sigma & = & \begin{cases} \mathbf{true}, \text{falls } [\![a_1]\!]\sigma \leq [\![a_2]\!]\sigma \\ \mathbf{false}, \text{sonst} \end{cases} \\
[\![\neg b]\!]\sigma & = & \begin{cases} \mathbf{true}, \text{falls } [\![b]\!]\sigma = \mathbf{false} \\ \mathbf{false}, \text{sonst} \end{cases} \\
[\![b_1{\wedge}b_2]\!]\sigma & = & \begin{cases} \mathbf{true}, \text{falls } [\![b_1]\!]\sigma = \mathbf{true} \text{ und } [\![b_2]\!]\sigma = \mathbf{true} \\ \mathbf{false}, \text{sonst} \end{cases}
\end{array}$$

Man beachte bei den arithmetischen Ausdrücken die doppelte Verwendung der Operatorsymbole $+,-,\cdot$. Auf der linken Seite handelt es sich um (abstrakte)

Syntax; auf der rechten um die Rechenoperationen mit der üblichen aus der Schule bekannten Bedeutung.

Man könnte den Ausdrücken auch eine operationale Semantik geben, etwa durch formale Beschreibung eines Rechenwerks (ALU). Die Übereinstimmung der operationalen und der denotationalen Semantik entspräche dann einem Korrektheitsbeweis für das Rechenwerk.

2.4 Operationale Semantik von IMP

Zur Festlegung der Semantik von Befehlen führen wir das Urteil $\sigma \xrightarrow{c} \sigma'$ ein. Es soll bedeuten, dass die Ausführung des Befehls c im Zustand σ den Zustand σ' herbeiführt. Dieses Urteil definieren wir induktiv durch folgende Regeln.

$$\text{(E-S{\scriptsize KIP})} \quad \frac{}{\sigma \xrightarrow{\textbf{skip}} \sigma}$$

$$\text{(E-A{\scriptsize SS})} \quad \frac{}{\sigma \xrightarrow{X:=a} \sigma[X \mapsto [\![a]\!]\sigma]}$$

$$\text{(E-S{\scriptsize EQ})} \quad \frac{\sigma \xrightarrow{c_1} \sigma' \quad \sigma' \xrightarrow{c_2} \sigma''}{\sigma \xrightarrow{c_1;c_2} \sigma''}$$

$$\text{(E-I{\scriptsize F}-T)} \quad \frac{\sigma \xrightarrow{c_1} \sigma'}{\sigma \xrightarrow{\textbf{if}\, b\, \textbf{then}\, c_1\, \textbf{else}\, c_2} \sigma'} \quad \text{falls } [\![b]\!]\sigma = \textbf{true}$$

$$\text{(E-I{\scriptsize F}-F)} \quad \frac{\sigma \xrightarrow{c_2} \sigma'}{\sigma \xrightarrow{\textbf{if}\, b\, \textbf{then}\, c_1\, \textbf{else}\, c_2} \sigma'} \quad \text{falls } [\![b]\!]\sigma = \textbf{false}$$

$$\text{(E-W{\scriptsize HILE}-F)} \quad \frac{}{\sigma \xrightarrow{\textbf{while}\, b\, \textbf{do}\, c} \sigma} \quad \text{falls } [\![b]\!]\sigma = \textbf{false}$$

$$\text{(E-W{\scriptsize HILE}-T)} \quad \frac{\sigma \xrightarrow{c} \sigma' \quad \sigma' \xrightarrow{\textbf{while}\, b\, \textbf{do}\, c} \sigma''}{\sigma \xrightarrow{\textbf{while}\, b\, \textbf{do}\, c} \sigma''} \quad \text{falls } [\![b]\!]\sigma = \textbf{true}$$

Wir schreiben jetzt $\sigma \xrightarrow{c} \sigma'$, wenn dieses Urteil mit Hilfe der obigen Regeln hergeleitet werden kann. Man beachte, dass die letzten vier Regeln mit Nebenbedingungen behaftet sind; sie sind nur anwendbar, wenn die Nebenbedingung erfüllt ist.

2.4.1 Beispiele

Wir betrachten wieder obiges Beispielprogramm c_{div} zur Berechung der ganzzahligen Division.

$$\begin{array}{l}
\texttt{A} := 0;\\
\texttt{B} := \texttt{X};\\
\textbf{if}\ \texttt{Y} \leq 0\ \textbf{then}\ \textbf{skip}\ \textbf{else}\\
\quad \textbf{while}\ \texttt{Y} \leq \texttt{B}\ \textbf{do}\\
\qquad \texttt{B} := \texttt{B} - \texttt{Y};\\
\qquad \texttt{A} := \texttt{A} + 1
\end{array}$$

Sei σ_0 ein beliebiger Zustand.

Der Initialisierungsteil des Programms wird in σ_0 so ausgewertet.

$$\dfrac{\dfrac{}{\sigma_0 \xrightarrow{\texttt{A:=0}} \sigma_0[\texttt{A}\mapsto 0]}\ \text{E-Ass} \qquad \dfrac{}{\sigma_0[\texttt{A}\mapsto 0] \xrightarrow{\texttt{B:=X}} \sigma_0[\texttt{A}\mapsto 0][\texttt{B}\mapsto\sigma_0(\texttt{X})]}\ \text{E-Ass}}{\sigma_0 \xrightarrow{\texttt{A:=0;B:=X}} \sigma_0[\texttt{A}\mapsto 0][\texttt{B}\mapsto\sigma_0(\texttt{X})]}\ \text{E-Seq}$$

Wir haben hier verwendet, dass $[\![0]\!]\sigma = 0$ und $[\![\texttt{X}]\!]\sigma_0 = \sigma_0(\texttt{X})$.

Wir schreiben jetzt $\sigma_1 \stackrel{\text{def}}{=} \sigma_0[\texttt{A}\mapsto 0][\texttt{B}\mapsto\sigma_0(\texttt{X})]$ für den nach der Initialisierung erreichten Zustand und weiter $b \stackrel{\text{def}}{=} \texttt{Y} \leq \texttt{B}$ und $c \stackrel{\text{def}}{=} \texttt{B} := \texttt{B} - \texttt{Y}; \texttt{A} := \texttt{A} + 1$ für Bedingung und Rumpf der **while**-Schleife unseres Programms.

Wir nehmen jetzt konkret an, dass $\sigma_0(\texttt{X}) = 5$ und $\sigma_0(\texttt{Y}) = 3$. Es ist daher $\sigma_1(\texttt{Y}) \leq \sigma_1(\texttt{B})$ und wir haben

$$\dfrac{\dfrac{\dfrac{\cdots}{\sigma_1 \xrightarrow{c} \sigma_0[\texttt{A}\mapsto 1][\texttt{B}\mapsto 2]} \qquad \dfrac{}{\sigma_1[\texttt{A}\mapsto 1][\texttt{B}\mapsto 2] \xrightarrow{\textbf{while}\, b\, \textbf{do}\, c} \sigma_0[\texttt{A}\mapsto 1][\texttt{B}\mapsto 2]}\ \text{E-While-F}}{\sigma_1 \xrightarrow{\textbf{while}\, b\, \textbf{do}\, c} \sigma_0[\texttt{A}\mapsto 1][\texttt{B}\mapsto 2]}\ \text{E-While-T}}{\sigma_1 \xrightarrow{\textbf{if}\ \texttt{Y}\leq 0\ \textbf{then}\,\textbf{skip}\ \textbf{else}\ \textbf{while}\, b\, \textbf{do}\, c} \sigma_0[\texttt{A}\mapsto 1][\texttt{B}\mapsto 2]}\ \text{E-If-F}$$

Es kommt also das richtige Resultat heraus. Wollen wir das für beliebigen Anfangszustand σ_0 zeigen, so müssen wir etwas allgemeiner vorgehen. Wir bezeichnen mit P die Menge aller Zustände σ für die gilt

$$\sigma(\texttt{A}) \cdot \sigma(\texttt{Y}) + \sigma(\texttt{B}) = \sigma(\texttt{X}) \wedge 0 \leq \sigma(\texttt{B})$$

Wir zeigen jetzt, dass P eine *Invariante* der Schleife ist, d.h. wenn $\sigma \xrightarrow{\texttt{B:=B-Y;A:A+1}} \tau$ und $[\![b]\!]\sigma = \textbf{true}$, so folgt aus $\sigma \in P$, dass auch $\tau \in P$. Wegen E-Seq und E-Ass gilt unter den Voraussetzungen $\tau(\texttt{A}) = \sigma(\texttt{A}) + 1$ und $\tau(\texttt{B}) = \sigma(\texttt{B}) - \sigma(\texttt{Y})$, sowie $\tau(\texttt{X}) = \sigma(\texttt{X})$ und $\tau(\texttt{Y}) = \sigma(\texttt{Y})$. Dann ist also

$$\begin{array}{rl}
 & \tau(\texttt{A}) \cdot \tau(\texttt{Y}) + \tau(\texttt{B})\\
= & (\sigma(\texttt{A}) + 1) \cdot \sigma(\texttt{Y}) + \sigma(\texttt{B}) - \sigma(\texttt{Y})\\
= & \sigma(\texttt{X}) = \tau(\texttt{X})
\end{array}$$

Wegen $[\![b]\!]\sigma = \mathbf{true}$ haben wir $0 \le \sigma(\mathtt{B}) - \sigma(\mathtt{Y}) = \tau(\mathtt{B})$. Also ist $\tau \in P$.

Wenn jetzt $\sigma \xrightarrow{\mathbf{while}\, b\, \mathbf{do}\, c} \tau$ und $\sigma \in P$, dann ist auch $\tau \in P$ und außerdem gilt $\tau(\mathtt{B}) < \tau(\mathtt{Y}) = \sigma(\mathtt{Y})$. Um das zu sehen machen wir eine Induktion über die Herleitung von $\sigma \xrightarrow{\mathbf{while}\, b\, \mathbf{do}\, c} \tau$, bzw. über die Anzahl der Durchläufe, was dasselbe ist. Wurde $\sigma \xrightarrow{\mathbf{while}\, b\, \mathbf{do}\, c} \tau$ mit E-WHILE-F erzielt, so ist $\sigma = \tau$ und $[\![b]\!]\sigma = \mathbf{false}$, also gilt die Behauptung. Wurde dagegen E-WHILE-T verwendet, so gilt $[\![b]\!]\sigma = \mathbf{true}$ und wir haben einen Zwischenzustand σ' mit $\sigma \xrightarrow{c} \sigma'$ und $\sigma' \xrightarrow{\mathbf{while}\, b\, \mathbf{do}\, c} \tau$. Nach obiger Beobachtung ist $\sigma' \in P$ und mit der Induktionsannahme gilt dann $\tau \in P$ und $\tau(\mathtt{B}) < \tau(\mathtt{Y}) = \sigma'(\mathtt{Y}) = \sigma(\mathtt{Y})$.

Wenn jetzt $\sigma_0(\mathtt{X}) \ge 0$, so ist $\sigma_1 \in P$. Ist außerdem $\sigma_0(\mathtt{Y}) > 0$, dann ergibt sich der Endzustand σ_2 aus $\sigma_1 \xrightarrow{\mathbf{while}\, b\, \mathbf{do}\, c} \sigma_2$ wegen E-IF-F. Existiert überhaupt so ein Zustand σ_2, terminiert also die **while**-Schleife, so gilt $\sigma_2 \in P$ und $\sigma_2(\mathtt{B}) < \sigma_2(\mathtt{Y})$.

Nach Ablauf unseres Programms stehen also in `A` und `B` tatsächlich Quotient und Rest der Division `X/Y` unter der Voraussetzung, dass $\mathtt{X} \ge 0$ und $\mathtt{Y} > 0$.

3 Axiomatische Semantik von IMP

Wir wollen jetzt die Vorgehensweise bei obigem Korrektheitsbeweis systematisieren und einen Satz von Schlussregeln angeben (den sog. Hoare Kalkül), der vollständig ist in dem Sinne, dass sich alle wahren Korrektheitsaussagen allein unter Verwendung dieser Schlussregeln gewinnen lassen.

Zunächst einmal müssen wir uns über die Form möglicher Korrektheitsaussagen klar werden. Man unterscheidet hier zwischen *partieller* und *totaler* Korrektheit. Unter einer partiellen Korrektheitsaussage über ein Programm c versteht man eine Aussage der Form

„Erfüllt ein Zustand σ die Eigenschaft P (Vorbedingung) und terminiert c im Zustand σ mit einem Endzustand τ (also $\sigma \xrightarrow{c} \tau$), so erfüllt τ die Eigenschaft Q (Nachbedingung).“

In unserem Beispiel war die Vorbedingung

$$P(\sigma) \overset{\text{def}}{\Leftrightarrow} \sigma(\mathtt{X}) \ge 0 \wedge \sigma(\mathtt{Y}) > 0$$

Die Nachbedingung war

$$Q(\tau) \overset{\text{def}}{\Leftrightarrow} \tau(\mathtt{A}) \cdot \tau(\mathtt{Y}) + \tau(\mathtt{B}) = \tau(\mathtt{X}) \wedge 0 \le \tau(\mathtt{B}) < \tau(\mathtt{Y})$$

In unserem informellen Beweis haben wir also die partielle Korrektheit des Programms c_{div} bezüglich dieser beiden Vor- und Nachbedingungen gezeigt.

So eine partielle Korrektheitsaussage hilft nur weiter, wenn man sich von der Termination des Programmes anderweitig überzeugt, denn durch ein nichtterminierendes Programm wird sie trivialerweise erfüllt.

Partielle Korrektheit zusammen mit Termination bezeichnet man als *totale Korrektheit.* Eine totale Korrektheitsaussage über ein Programm c ist also eine Aussage der Form

„Erfüllt ein Zustand σ die Eigenschaft P, so terminiert die Ausführung von c in einem (notwendigerweise eindeutigen) Zustand τ, der die Eigenschaft Q (Nachbedingung) hat.“

3.1 Zusicherungen

Die Eigenschaften P und Q bezeichnet man als *Zusicherungen*, engl. *assertions.* Über die Form dieser Zusicherungen wollen wir uns jetzt Gedanken machen. Seien P und Q die beiden Zusicherungen aus dem Beispiel. Die Aufgabe, ein Programm c zu finden, das bzgl. der Vor- und Nachbedingungen P, Q aus dem Beispiel partiell oder sogar total korrekt ist, lässt sich folgendermaßen auf triviale Weise lösen

$$\mathtt{X} := 0; \mathtt{Y} := 1; \mathtt{A} := 0; \mathtt{B} := 0$$

Der Grund ist, dass dieses Programm seine Eingabe abändert. Möchte man so etwas ausschließen, dann muss man die Möglichkeit schaffen, bestimmte Variablenwerte festzuhalten. Im Beispiel können wir das dadurch erreichen, dass wir die Vor- und Nachbedingung abändern zu

$$P'(\sigma) \overset{\text{def}}{\Leftrightarrow} P(\sigma) \wedge \sigma(\mathtt{X}) = x \wedge \sigma(\mathtt{Y}) = y$$
$$Q'(\tau) \overset{\text{def}}{\Leftrightarrow} Q(\tau) \wedge \tau(\mathtt{X}) = x \wedge \tau(\mathtt{Y}) = y$$

Die beiden Buchstaben x und y sind hier sog. *Hilfsvariablen*, engl. *auxiliary variables.*

Diese Hilfsvariablen formalisieren wir durch einen ein für allemal festgelegten Parameterraum Π. In unserem Beispiel kann man für den Parameterraum die Menge der Funktionen von $\{x, y\}$ nach **int** nehmen.

Definition 3.1 *Eine* Zusicherung *ist eine Relation zwischen Zuständen und Parametern, also formal eine Teilmenge von* $\Sigma \times \Pi$.

Im Beispiel wäre etwa

$$P'(\sigma, \pi) \overset{\text{def}}{\Leftrightarrow} P(\sigma) \wedge \sigma(\mathtt{X}) = \pi(x) \wedge \sigma(\mathtt{Y}) = \pi(y)$$

Im allgemeinen reicht es aus, für den Parameterraum die Menge Σ der Zustände zu nehmen; damit hat man für jede Programmvariable eine Hilfsvariable, um ihren Wert zu fixieren. Wir lassen aber doch den Parameterraum beliebig.

Definition 3.2 *Ein* Hoare-Tripel *ist ein Ausdruck der Form* $\{P\}c\{Q\}$*, wobei* P, Q *Zusicherungen sind und* c *ein Programm ist. Die Zusicherung* P *heißt Vorbedingung, die Zusicherung* Q *heißt Nachbedingung des Tripels. Wir sagen* $\{P\}c\{Q\}$ gilt bzgl. partieller Korrektheit *und schreiben*

$$\models_p \{P\}c\{Q\}$$

wenn gilt

$$\forall \sigma \in \Sigma . \forall \pi \in \Pi . P(\sigma, \pi) \Rightarrow \forall \tau \in \Sigma . \sigma \xrightarrow{c} \tau \Rightarrow Q(\tau, \pi)$$

Wir sagen $\{P\}c\{Q\}$ *gilt bzgl. totaler Korrektheit und schreiben*

$$\models_t \{P\}c\{Q\}$$

wenn gilt

$$\forall \sigma \in \Sigma . \forall \pi \in \Pi . P(\sigma, \pi) \Rightarrow \exists \tau \in \Sigma . \sigma \xrightarrow{c} \tau \wedge Q(\tau, \pi)$$

Wir sagen auch $\{P\}c\{Q\}$ gilt und schreiben dafür $\models \{P\}c\{Q\}$, wenn aus dem Zusammenhang klar ist, welcher Korrektheitsbegriff gemeint ist.

Für partielle Korrektheit spricht, dass die zugehörigen Beweismethoden einfacher sind und dass die Termination oft unmittelbar klar ist (z.B. wenn nur *for*-Schleifen verwendet werden) oder leicht separat gezeigt werden kann (meist durch Angabe eines $\mathbf{N}$-wertigen Ausdrucks, dessen Wert sich bei jedem Schleifendurchlauf reduziert.

Für totale Korrektheit spricht, dass manchmal eben der Terminationsbeweis doch nicht einfach entkoppelt werden kann, z.B. wenn die Reduktion des Terminationsmaßes vom Erfülltsein bestimmter lokaler Bedingungen abhängt. Auch kann man argumetieren, dass *wenn* der Beweis der Termination tatsächlich einfach ist, dessen formale Hinzunahme auch nicht allzuviel ausmacht.

Im Rahmen dieser Vorlesung werden wir in erster Linie aber doch die partielle Korrektheit betrachten, da die zugehörige Theorie etwas einfacher ist. Auf nötige Änderung für totale Korrektheit werde ich von Fall zu Fall hinweisen.

3.1.1 VDM

Im Prinzip kann man auf Hilfsvariablen in der Vorbedingung verzichten, wenn man festlegt, dass zu jeder Programmvariablen in der Nachbedingung eine Hilfsvariable zur Verfügung steht, die an den Wert der Variablen vor der Ausführung des Programms gebunden wird. Mit anderen Worten, wenn $P \subseteq \Sigma$ und $Q \subseteq \Sigma \times \Sigma$, so können wir die partielle Korrektheit von c bzgl. P und Q formulieren als

$$\forall \sigma . P(\sigma) \Rightarrow \forall \tau . \sigma \xrightarrow{c} \tau \Rightarrow Q(\sigma, \tau)$$

Analog dazu die totale Korrektheit. Dieser Ansatz liegt der *Vienna Development Method* (VDM) zugrunde, einer der auch industriell mit Erfolg angewandten Ansätze zur formalen Programmverifikation. Ein Vorteil der allgemeineren Fassung liegt darin, dass man dann bei Bedarf auch auf noch frühere Werte einer Variablen zurückgreifen kann.

3.1.2 Notation für Zusicherungen

Ist der Parameterraum Π von der Form $\mathbf{Avar} \to \mathbf{int}$, wobei **Avar** eine von **var** disjunkte Menge von Hilfsvariablen ist, so schreiben wir bei der Definition einer Zusicherung einfach X für $\sigma(X)$ und x für $\pi(x)$, falls $X \in \mathbf{var}$ und $x \in \mathbf{Avar}$. In den Beispielen nehmen wir für die Hilfsvariablen kleine *schräggedruckte* Buchstaben.

Zum Beispiel könnten wir die Zusicherungen aus dem Divisionsprogramm hinschreiben als

$$P' \overset{\text{def}}{\Leftrightarrow} \mathtt{X} \geq 0 \wedge \mathtt{Y} > 0 \wedge \mathtt{X} = x \wedge \mathtt{Y} = y$$
$$Q' \overset{\text{def}}{\Leftrightarrow} \mathtt{X} = x \wedge \mathtt{Y} = y \wedge \mathtt{A} \cdot \mathtt{Y} + \mathtt{B} = \mathtt{X} \wedge \mathtt{B} < \mathtt{Y}$$

Zu jedem booleschen Ausdruck können wir eine Zusicherung P_b definieren durch

$$P_b(\sigma, \pi) \overset{\text{def}}{\Leftrightarrow} [\![b]\!]\sigma = \mathbf{true}$$

Abkürzend schreiben wir b statt P_b. Da wir für boole'sche Ausdrücke dieselbe Syntax, wie für die Metalogik verwenden, ist diese Konvention für konkrete Beispiele schon durch die Kurzschreibweise X statt $\sigma(X)$ abgedeckt.

In der Tat waren alle bisher vorkommenden Zusicherungen von dieser Form. Das muß aber nicht so sein. Ist etwa c ein Programm, dann können wir die Zusicherung

$$P(\sigma, \pi) \overset{\text{def}}{\Leftrightarrow} \exists \tau . \sigma \xrightarrow{c} \tau$$

betrachten. Sie bezeichnet die Menge aller der Zustände von denen aus c terminiert. Andere Beispiele sind Zusicherungen wie „X ist prim“ oder „X und Y sind teilerfremd“.

Ist P eine Zusicherung, X eine Programmvariable und $a \in \mathbf{aexp}$, so schreiben wir $P[X \mapsto a]$ für die Zusicherung, die man aus P erhält, indem man für X den Ausdruck a einsetzt. Formal bedeutet das

$$P[X \mapsto a](\sigma, \pi) \overset{\text{def}}{\Leftrightarrow} P(\sigma[X \mapsto [\![a]\!]\sigma], \pi)$$

Oft werden diese Notationen formalisiert und man definiert Zusicherungen als gewisse syntaktische Ausdrücke, deren Semantik dann eine Relation zwischen Zuständen und Parametern ist. Das ist wichtig, wenn man Äquivalenz und Implikation zwischen Zusicherungen in einem formalen System statt furch informelles Argumentieren beweisen möchte.

3.2 Der Hoare-Kalkül

Wir geben jetzt einen Satz von logischen Schlußregeln an, die es uns gestatten, gültige *partielle* Korrektheitsaussagen der Form $\{P\}c\{Q\}$ formal herzuleiten. Für jedes programmiersprachliche Konstrukt gibt es eine Regel, die Anwendung findet, wenn dieses Konstrukt ganz außen steht. Beginnen wir mit der leeren Anweisung.

$$\text{(H-S{\scriptsize KIP})} \quad \frac{}{\{P\}\mathbf{skip}\{P\}}$$

Gilt also eine Zusicherung vor Ausführung der leeren Anweisung, dann gilt sie hinterher auch. Etwas interessanter ist schon die Regel für die sequentielle Komposition.

$$\text{(H-S{\scriptsize EQ})} \quad \frac{\{P\}c_1\{P'\} \quad \{P'\}c_2\{Q\}}{\{P\}c_1; c_2\{Q\}}$$

Wendet man diese Regel rückwärts an, so muss man die Zusicherung P' „raten“; wir werden aber sehen, dass man meistens eine kanonische Wahl treffen kann.

Die Regel für die Wertzuweisung basiert auf der Substitution.

$$\text{(H-A{\scriptsize SS})} \quad \frac{}{\{P[X \mapsto a]\}X{:=}a\{P\}}$$

Gilt $P[X \mapsto a]$ vor einer Wertzuweisung, so gilt P *ohne* Substitution nach der Wertzuweisung, da dann ja $[\![X]\!]\sigma = [\![a]\!]\sigma$ für alle σ.

Bei der Fallunterscheidung machen wir von der Möglichkeit Gebrauch, boole'sche Ausdrücke als Zusicherungen aufzufassen.

$$\text{(H-I{\scriptsize F})} \quad \frac{\{P \wedge b\}c_1\{Q\} \quad \{P \wedge \neg b\}c_2\{Q\}}{\{P\}\mathbf{if}\, b\, \mathbf{then}\, c_1\, \mathbf{else}\, c_2\{Q\}}$$

Wenn also unter Annahme von b und P nach c_1 die Zusicherung Q gilt und unter Annahme von $\neg b$ und P nach c_2 die Zusicherung Q gilt, dann gilt Q nach $\mathbf{if}\, b\, \mathbf{then}\, c_1\, \mathbf{else}\, c_2$ unter Annahme von P alleine.

Schließlich die Regel für die **while**-Schleife. Beachte, dass wir im Moment partielle Korrektheit betrachten.

$$\text{(H-W{\scriptsize HILE})} \quad \frac{\{P \wedge b\}c\{P\}}{\{P\}\mathbf{while}\, b\, \mathbf{do}\, c\{P \wedge \neg b\}}$$

Ist also unter der Zusatzannahme b die Zusicherung P eine Invariante der Schleife, so gilt P nach Ausführung der Schleife unter Voraussetzung der Termination, falls P am Anfang gegolten hat; außerdem wissen wir nach Beendigung der Schleife noch, dass b nicht mehr gilt; sonst hätten wir die Schleife ja nicht verlassen.

Diese Regeln sind nun leider noch nicht ganz ausreichend; denn mit ihnen lässt sich zum Beispiel eine Aussage der Form $\{P\}\mathbf{while}\, b\, \mathbf{do}\, c\{Q\}$ nur dann zeigen, wenn Q die Form $P \wedge \neg b$ hat, was im allgemeinen und auch in unserem Beispiel ja nicht der Fall ist. Außerdem beinhaltet keine dieser Regeln echtes mathematisches Argumentieren; wären die Regeln ausreichend, dann würde sich Programmverifikation auf syntaxgerichtetes Rückwärtsanwenden von Regeln beschränken. Um dem abzuhelfen, führen wir noch folgende Konsequenzregel ein:

$$(\text{H-C{\small ONS}}) \quad \frac{P \subseteq P' \quad \{P'\}c\{Q'\} \quad Q' \subseteq Q}{\{P\}c\{Q\}}$$

Die beiden Nebenbedingungen $P \subseteq P'$ und $Q' \subseteq Q$ drücken aus, dass P' schwächer ist als P und umgekehrt Q' stärker als Q. Die Konsequenzregel erlaubt es uns also, zum Beweis einer Korrektheitsaussage die Vorbedingung abzuschwächen und die Nachbedingung zu verstärken. Auf den ersten Blick sieht es nicht so aus, als könnte man dadurch etwas gewinnen. Der Punkt ist, dass man zum Beipiel durch Rückwärtsanwendung der Konsequenzregel die Situation für die Anwendung der **while**-Regel oder der Zuweisungsregel herbeiführen kann.

Durch die Konsequenzregel wird der Hoare-Kalkül zu einem semi-formalen System; möchte man die Regel anwenden, so ist ein echter mathematischer Beweis gefragt, dass die beiden Nebenbedingungen $P \subseteq P'$ und $Q' \subseteq Q$ erfüllt sind. Solch ein Beweis kann beliebig kompliziert sein. Man kann es ja zum Beispiel so einrichten, dass P der Menge der Gegenbeispiele zu einem schwierigen Theorem ist und $P' = \emptyset$. Dann ist $P \subseteq P'$ äquivalent zur Gültigkeit dieses Theorems. Solche Situationen können in der Programmverifikation wirklich vorkommen. Betrachte zum Beispiel ein Programm c, welches systematisch nach einem Gegenbeispiel zu Fermat-Wiles' Theorem unterhalb des Wertes der Programmvariablen `X` sucht. Hat es ein solches gefunden, so setzt c die Variable `Y` auf 0, anderenfalls auf 1. Die partielle (oder totale) Korrektheitsaussage $\{\mathtt{X} \geq 0\}c\{\mathtt{Y} = 0\}$ ist in diesem Fall äquivalent zu Fermat-Wiles' Theorem.

Manche Autoren (z.B. Mitchell) sehen darin ein Argument gegen formale Programmverifikation. Dazu ist zu sagen, dass die meisten praktisch vorkommenden Programme entweder einfache Korrektheitsbeweise haben, oder aber sich auf *Algorithmen* abstützen, deren Korrektheit man voraussetzt. Das Problem bei der Programmverifikation ist auch nicht die mathematische Tiefe der erforderlichen Beweise, sondern vielmehr deren Unübersichtlichkeit. Diese aber erhofft man durch Einsatz formaler Methoden, ggf. mit Computerunterstützung in den Griff zu bekommen.

3.2.1 Beispiel

Hier ist ein formaler Beweis der partiellen Korrektheit des Hauptteils unseres Beispielprogramms im Hoare-Kalkül. Wir schreiben

$$P \overset{\text{def}}{\Leftrightarrow} \mathtt{X}{=}x \wedge \mathtt{Y}{=}y \wedge \mathtt{X}{\geq}0 \wedge \mathtt{Y}{>}0$$

als Abkürzung für die Bedingungen an die Eingabevariablen.

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{}{\begin{array}{l}\{P \land (\mathtt{A}+1)\cdot\mathtt{Y}+\mathtt{B}-\mathtt{Y}=\mathtt{X} \land \mathtt{Y} \leq \mathtt{B}\}\\ \quad \mathtt{A}:=\mathtt{A}+1\\ \{P \land \mathtt{A}\cdot\mathtt{Y}+\mathtt{B}-\mathtt{Y}=\mathtt{X} \land \mathtt{Y} \leq \mathtt{B}\}\end{array}}\,\text{H-Ass}
\qquad
\dfrac{}{\begin{array}{l}\{P \land \mathtt{A}\cdot\mathtt{Y}+\mathtt{B}-\mathtt{Y}=\mathtt{X} \land 0 \leq \mathtt{B}-\mathtt{Y}\}\\ \quad \mathtt{B}:=\mathtt{B}-\mathtt{Y}\\ \{P \land \mathtt{A}\cdot\mathtt{Y}+\mathtt{B}=\mathtt{X} \land 0\leq\mathtt{B}\}\end{array}}\,\text{H-Ass}
}{\begin{array}{l}\{P \land \mathtt{A}\cdot\mathtt{Y}+\mathtt{B}=\mathtt{X} \land 0\leq\mathtt{B} \land \mathtt{Y} \leq \mathtt{B}\}\\ \quad \mathtt{A}:=\mathtt{A}+1;\, \mathtt{B}:=\mathtt{B}-\mathtt{Y}\\ \{P \land \mathtt{A}\cdot\mathtt{Y}+\mathtt{B}=\mathtt{X} \land 0\leq\mathtt{B}\}\end{array}}\,\text{H-Seq}
}{\begin{array}{l}\{P \land \mathtt{A}\cdot\mathtt{Y}+\mathtt{B}=\mathtt{X} \land 0\leq\mathtt{B}\}\\ \quad \textbf{while}\ \mathtt{Y}\leq\mathtt{B}\ \textbf{do}\ (\mathtt{A}:=\mathtt{A}+1;\, \mathtt{B}:=\mathtt{B}-\mathtt{Y})\\ \{P \land \mathtt{A}\cdot\mathtt{Y}+\mathtt{B}=\mathtt{X} \land 0\leq\mathtt{B} \land \neg(\mathtt{Y}\leq\mathtt{B}\}\end{array}}\,\text{H-While}
}{\begin{array}{l}\{P \land \mathtt{A}=0 \land \mathtt{B}=\mathtt{X}\}\\ \quad \textbf{while}\ \mathtt{Y}\leq\mathtt{B}\ \textbf{do}\ (\mathtt{A}:=\mathtt{A}+1;\, \mathtt{B}:=\mathtt{B}-\mathtt{Y})\\ \{P \land \mathtt{A}\cdot\mathtt{Y}+\mathtt{B}=\mathtt{X} \land 0\leq\mathtt{B}<\mathtt{Y}\}\end{array}}\,\text{H-Cons}
$$

Man beachte, dass wir hier stellenweise Zusicherungen äquivalent umgeformt haben. Das ist erlaubt, da wir Zusicherungen als Relationen und nicht als Ausdrücke für Relationen auffassen. Hätten wir eine formale Syntax für Zusicherungen eingeführt, dann wären hier jeweils noch Instanzen der Konsequenzregel erforderlich gewesen. Die letzte (explizit) angegebene Instanz der Konsequenzregel ist aber wirklich nötig, da wir die Vorbedingung echt abschwächen.

3.3 Korrektheit des Hoare-Kalküls

Wir wollen uns jetzt davon überzeugen, dass der Hoare-Kalkül nur wahre Korrektheitsaussagen herzuleiten gestattet. Führen wir die Notation $\vdash \{P\}c\{Q\}$ als Abkürzung dafür ein, daß das Hoare Tripel $\{P\}c\{Q\}$ mit Hilfe der obigen Regeln hergeleitet werden kann. Die Korrektheit des Hoare-Kalküls können wir dann folgendermaßen formulieren.

Satz 3.3 *Seien P, Q Zusicherungen und c ein Programm. Falls $\vdash \{P\}c\{Q\}$, so gilt $\models_p \{P\}c\{Q\}$.*

Den Beweis dieser Aussage führen wir durch Induktion über die (Länge der) Herleitung von $\vdash \{P\}c\{Q\}$.

Zunächst behandeln wir die beiden „Axiome“ H-Skip und H-Ass. Im Falle von H-Skip ist $P = Q$ und $c = \textbf{skip}$ und wir haben zu zeigen $\models \{P\}\textbf{skip}\{P\}$. Ausgeschrieben bedeutet das aber

$$\forall \sigma, \pi. P(\sigma, \pi) \Rightarrow \forall \tau. \sigma \xrightarrow{\textbf{skip}} \tau \Rightarrow P(\tau, \pi)$$

Geben wir uns also σ, π, τ mit $P(\sigma, \pi)$ und $\sigma \xrightarrow{\textbf{skip}} \tau$ vor. Aufgrund der Definition der operationalen Semantik muss gelten $\sigma = \tau$ und somit $P(\tau, \pi)$ nach Annahme.

Im Falle von H-Ass ist $c = X{:=}a$ und $P = Q[X \mapsto a]$. Wir haben zu zeigen $\models \{Q[X \mapsto a]\}X{:=}a\{Q\}$. Seien σ, π vorgegeben mit $P(\sigma, \pi)$, also $Q(\sigma[X \mapsto [\![a]\!]\sigma], \pi)$. Aus $\sigma \xrightarrow{X:=a} \tau$ folgt aber $\tau = \sigma[X \mapsto [\![a]\!]\sigma]$. Also gilt $Q(\tau, \pi)$.

Kommen wir zu den Schlußregeln H-Seq— H-While.

War die letzte Regel H-Seq, so ist $c = c_1; c_2$ und wir haben eine Zwischenzusicherung P', sodass gilt $\vdash \{P\}c_1\{P'\}$ und $\vdash \{P'\}c_2\{Q\}$. Nach Induktionsannahme gilt $\models \{P\}c_1\{P'\}$ und $\models \{P'\}c_2\{Q\}$. Seien nun σ, π mit $P(\sigma, \pi)$ gegeben und gelte $\sigma \xrightarrow{c_1;c_2} \tau$. Nach Definition der Auswerterelation muß es dann einen Zustand σ' geben mit $\sigma \xrightarrow{c_1} \sigma'$ und $\sigma' \xrightarrow{c_2} \tau$. Wegen $\models \{P\}c_1\{P'\}$ gilt dann $P'(\sigma', \pi)$ und weiter wegen $\models \{P'\}c_2\{Q\}$ schließlich $Q(\tau, \pi)$ w.z.b.w.

War die letzte Regel H-If, so ist $c = \mathbf{if}\, b\, \mathbf{then}\, c_1\, \mathbf{else}\, c_2$ und wir dürfen mit Induktion annehmen, daß $\models \{P \wedge b\}c_1\{Q\}$ und $\models \{P \wedge \neg b\}c_2\{Q\}$. Sind σ, π, τ mit $P(\sigma, \pi)$ und $\sigma \xrightarrow{c} \tau$ gegeben, so unterscheiden wir, ob $[\![b]\!]\sigma = \mathbf{true}$ oder $[\![b]\!]\sigma = \mathbf{false}$. Im ersten Fall gilt $\sigma \xrightarrow{c_1} \tau$ und wegen $(P \wedge b)(\sigma, \pi)$ nach Induktionsannahme $Q(\tau, \pi)$. Im zweiten Fall gilt $\sigma \xrightarrow{c_2} \tau$ und wir erhalten $Q(\tau, \pi)$ mit dem zweiten Teil der Induktionsvoraussetzung. In jedem Fall gilt also $Q(\tau, \pi)$.

War die letzte Regel H-While, so ist $c = \mathbf{while}\, b\, \mathbf{do}\, c_1$ und es gilt $Q = P \wedge \neg b$. Die Induktionsvoraussetzung lautet $\models \{P \wedge b\}c\{P\}$. Wir zeigen jetzt durch Induktion über Herleitungen in der operationalen Semantik, dass aus $\sigma \xrightarrow{\mathbf{while}\, b\, \mathbf{do}\, c_1} \tau$ und $P(\sigma, \pi)$ folgt $Q(\tau, \pi)$. Wurde $\sigma \xrightarrow{c} \tau$ mit E-While-F gewonnen, so ist $\sigma = \tau$ und $[\![b]\!]\sigma = \mathbf{false}$. Also folgt aus $P(\sigma, \pi)$ dann $(P \wedge \neg b)(\sigma, \pi)$, also $Q(\tau, \pi)$. Wurde $\sigma \xrightarrow{c} \tau$ dagegen mit E-While-T gewonnen, so haben ist $[\![b]\!]\sigma = \mathbf{true}$ und wir haben $\sigma \xrightarrow{c_1} \sigma'$ und $\sigma' \xrightarrow{c} \tau$. Wenn $P(\sigma, \pi)$, dann auch $(P \wedge b)(\sigma, \pi)$, also gemäß der Induktionsvoraussetzung unseres Korrektheitsbeweises $P(\sigma', \pi)$. Die lokale Induktionsvoraussetzung liefert dann $Q(\tau, \pi)$. $\square$

3.4 Vollständigkeit des Hoare-Kalküls

Wir haben gesehen, dass der Hoare-Kalkül nur wahre Aussagen herzuleiten gestattet; es fragt sich aber, ob wir durch Zwischenschalten der Abstraktionsebene des Hoare-Kalküls nicht zuviel Information verloren haben; mit anderen Worten, ob sich alle wahren partiellen Korrektheitsaussagen allein unter Zuhilfenahme der Schlußregeln gewinnen lassen, oder ob wir stellenweise doch die Definition der Hoare-Tripel und der operationellen Semantik heranziehen müssen. Die Antwort ist, daß die Schlußregeln tatsächlich ausreichen:

Satz 3.4 *Seien P, Q Zusicherungen und c ein Programm. Falls $\models_p \{P\}c\{Q\}$, so existiert eine Herleitung von $\{P\}c\{Q\}$, also $\vdash \{P\}c\{Q\}$.*

Zum Beweis führen wir den Begriff der *schwächsten Vorbedingung* ein.

Definition 3.5 *Sei Q eine Zusicherung und c ein Programm. Die* schwächste Vorbedingung *unter c für Q ist die durch*

$$\mathbf{wp}(c, Q)(\sigma, \pi) \overset{\text{def}}{\Leftrightarrow} \forall \tau.(\sigma \xrightarrow{c} \tau) \Rightarrow Q(\tau, \pi)$$

erklärte Zusicherung.

Die Bezeichnung „schwächste Vorbedingung" begründet sich aus folgenden zwei unmittelbar einsichtigen Eigenschaften.

Lemma 3.6 *Sei c Programm und Q Zusicherung.*

1. $\models_p \{\mathbf{wp}(c, Q)\}c\{Q\}$,
2. $\models_p \{P\}c\{Q\}$ *gilt genau dann, wenn* $P \subseteq \mathbf{wp}(c, Q)$, *d.h. wenn* $\forall \sigma, \pi. P(\sigma, \pi) \Rightarrow \mathbf{wp}(c, Q)(\sigma, \pi)$.

Ist also der Hoare-Kalkül tatsächlich vollständig, so muß wegen 3.6(1) auf jeden Fall gelten $\vdash \{\mathbf{wp}(c, Q)\}c\{Q\}$ für alle c und Q. Andererseits folgt dann aus 3.6(2) aus schon die Vollständigkeit mit einmaliger Anwendung der Konsequenzregel. Wir müssen also zum Beweis der Vollständigkeit folgendes zeigen:

Lemma 3.7 *Ist c Programm und Q Zusicherung, so gilt $\vdash \{\mathbf{wp}(c, Q)\}c\{Q\}$.*

Den Beweis führen wir durch Induktion über den Aufbau von c.

Fall $c = \mathbf{skip}$: Es ist

$$\begin{aligned} & \mathbf{wp}(c, Q)(\sigma, \pi) \\ \Leftrightarrow\ & \forall \tau. \sigma \xrightarrow{\mathbf{skip}} \tau \Rightarrow Q(\tau, \pi) \\ \Leftrightarrow\ & Q(\tau, \pi) \end{aligned}$$

Somit gilt $\vdash \{\mathbf{wp}(c, Q)\}c\{Q\}$ mit H-Skip.

Fall $c = X{:=}a$: Wenn $\sigma \xrightarrow{X:=a} \tau$, so ist $\tau = \sigma[X \mapsto [\![a]\!]\sigma]$. Daher ist $\mathbf{wp}(c, Q) \Leftrightarrow Q[X \mapsto a]$ und die Behauptung folgt mit H-Ass.

Fall $c = c_1; c_2$: Die Induktionshypothese liefert uns $\vdash \{\mathbf{wp}(c_2, Q)\}c_2\{Q\}$ und $\vdash \{\mathbf{wp}(c_1, (\mathbf{wp}(c_2, Q)))\}c_1\{\mathbf{wp}(c_2, Q)\}$, also $\vdash \{\mathbf{wp}(c_1, (\mathbf{wp}(c_2, Q)))\}c_1; c_2\{Q\}$ mit H-Seq. Es bleibt zu zeigen, daß $\mathbf{wp}(c_1, (\mathbf{wp}(c_2, Q))) = \mathbf{wp}(c_1; c_2, Q)$. Das sehen wir so.

$$
\begin{array}{ll}
& \mathbf{wp}(c_1; c_2, Q)(\sigma, \pi) \\
\Leftrightarrow & \forall \tau . \sigma \xrightarrow{c_1;c_2} \tau \Rightarrow Q(\tau, \pi)) \\
\Leftrightarrow & \forall \sigma', \tau . \sigma \xrightarrow{c_1} \sigma' \wedge \sigma' \xrightarrow{c_2} \tau \Rightarrow Q(\tau, \pi) \\
\Leftrightarrow & \forall \sigma' . \sigma \xrightarrow{c_1} \sigma' \Rightarrow \mathbf{wp}(c_2, Q)(\sigma', \pi) \\
\Leftrightarrow & \mathbf{wp}(c_1, (\mathbf{wp}(c_2, Q)))
\end{array}
$$

Fall $c = \mathbf{if}\, b\, \mathbf{then}\, c_1\, \mathbf{else}\, c_2$: Es ist

$$\mathbf{wp}(c, Q) = (b \Rightarrow \mathbf{wp}(c_1, Q)) \wedge (\neg b \Rightarrow \mathbf{wp}(c_2, Q))$$

Die Induktionsannahme liefert $\vdash \{\mathbf{wp}(c_1, Q)\}c_1\{Q\}$, also $\vdash \{b \wedge \mathbf{wp}(c, Q)\}c_1\{Q\}$. Analog erhalten wir $\vdash \{\neg b \wedge \mathbf{wp}(c, Q)\}c_2\{Q\}$. Also erhalten wir $\vdash \{\mathbf{wp}(c, Q)\}c\{Q\}$ mit H-If.

Fall $c = \mathbf{while}\, b\, \mathbf{do}\, c_1$: Es gilt

$$\mathbf{wp}(c, Q) = (\neg b \Rightarrow Q) \wedge (b \Rightarrow \mathbf{wp}(c_1, \mathbf{wp}(c, Q)))$$

Somit ist

$$b \wedge \mathbf{wp}(c, Q) = \mathbf{wp}(c_1, \mathbf{wp}(c, Q))$$

und die Induktionsvoraussetzung liefert $\vdash \{b \wedge \mathbf{wp}(c, Q)\}c_1\{\mathbf{wp}(c, Q)\}$. Die Regel H-While liefert dann $\vdash \{\mathbf{wp}(c, Q)\}c\{\neg b \wedge \mathbf{wp}(c, Q)\}$, also $\vdash \{\mathbf{wp}(c, Q)\}c\{Q\}$. □

Fassen wir zusammen: Wenn $\models_p \{P\}c\{Q\}$, so folgt aus der Definition von $\mathbf{wp}(c, Q)$ und $\models_p$ daß $P \subseteq \mathbf{wp}(c, Q)$. Andererseits liefert uns das eben gezeigte Lemma einen Beweis von $\vdash \{\mathbf{wp}(c, Q)\}c\{Q\}$. Eine Anwendung von H-Cons ergibt $\vdash \{P\}c\{Q\}$.

3.5 Verifikationsbedingungen

Dieser Vollständigkeitsbeweis lehrt uns zwei Dinge. Erstens liefert die schwächste Vorbedingung einen kanonischen Vorschlag für die Zwischenzusicherung bei der

Konsequenzregel: Ist $\{P\}c_1; c_2\{Q\}$ zu zeigen, so vergibt man sich nichts, wenn man versucht, $\{P\}c_1\{\mathbf{wp}(c_2, Q)\}$ zu zeigen ($\{P\}c_1; c_2\{Q\}$ folgt dann mit Lemma 3.7 und H-Seq). Diese Strategie ist insbesondere dann gut, wenn man die schwächste Vorbedingung leicht explizit angeben kann, z.B. wenn c_2 eine Wertzuweisung ist.

Zweitens zeigt uns der Beweis, dass wir für jede **while**-Schleife **while** b **do** c eine Invariante P finden können, welche zusammen mit $\neg b$ eine beliebige Nachbedingung Q impliziert. Man setzt nämlich $P = \mathbf{wp}(\mathbf{while}\, b\, \mathbf{do}\, c, Q)$. Man kann diese schwächste Vorbedingung auch explizit angeben: Es gilt nämlich $\mathbf{wp}(\mathbf{while}\, b\, \mathbf{do}\, c, Q)(\sigma, \pi)$ genau dann, wenn für jede Zustandsfolge $\sigma_1, \dots, \sigma_n$ mit $\sigma_i \xrightarrow{c} \sigma_{i+1}$ und $[\![b]\!]\sigma_i = \mathbf{false}$ für $1 \leq i < n$ und $[\![b]\!]\sigma_n = \mathbf{true}$ gilt $Q(\sigma_n, \pi)$. Klar, dass so eine Zustandsfolge eindeutig bestimmt ist, falls sie überhaupt existiert.

Leider hilft diese Art von Invariante nicht viel weiter, da man mit dieser Wahl die gesamte Schwierigkeit der Programmverifikation auf die Nebenbedingungen der Konsequenzregel verlagert. In vielen praktisch vorkommenden Fällen hat man jedoch schon beim Entwurf des Programms eine geeignete Invariante vor Augen, bei deren Verwendung sich die Nebenbedingungen zu einfachen arithmetischen Aussagen reduzieren. Die Angabe dieser Invarianten ist aber auch fast der einzige „Grips“, der in einem Beweis im Hoare-Kalkül steckt.

Wir wollen im folgenden ein Verfahren betrachten, welches aus einem mit Invarianten annotierten Programm automatisch sog. *Verifikationsbedingungen* generiert, deren Wahrheit die partielle Korrektheit des Programms nach sich zieht. Außerdem liefert das Verfahren dann einen Beweisbaum für die Korrektheit des Programms, in welchem die Verifikationsbedingungen als Nebenbedingungen der Konsequenzregel vorkommen und in welchem bei der H-While-Regel jeweils die vorgegebene Invariante verwendet wird.

Die Idee ist es dann, die Invarianten mit menschlicher Intuition so geschickt zu wählen, dass die Verifikationsbedingungen möglichst automatisch durch einen Theorembeweiser gezeigt werden können.

Ein *annotiertes Programm* ist ein Programm, in welchem jede **while**-Schleife mit einer Zusicherung versehen ist. Formal kann man die annotierten Programme durch folgende Grammatik erzeugen, wobei S eine Zusicherung ist:

$$c ::= \mathbf{skip} \mid X{:=}a \mid c_1; c_2 \mid \mathbf{if}\, b\, \mathbf{then}\, c_1\, \mathbf{else}\, c_2 \mid \mathbf{while}_{\{S\}}\, b\, \mathbf{do}\, c$$

Die operationale Semantik von annotierten Programmen ist genau wie die der nicht annotierten. Zweck der Annotate ist es, gerade soviel menschliche Hilfestellung zu geben, wie nötig ist, um einen Beweisskizze im Hoare-Kalkül zu erzeugen. Für jedes Hoare-Tripel $\{P\}c\{Q\}$ wobei c ein annotiertes Programm ist, definieren wir jetzt eine Menge von Aussagen $\mathbf{vc}(\{P\}c\{Q\})$ deren Wahrheit $\models_p \{P\}c\{Q\}$ impliziert und außerdem einen Beweis im Hoare-Kalkül liefert, welcher jeweils die vorgegebenen Invarianten bei der Regel H-While verwendet. Die Aussagen in $\mathbf{vc}(\{P\}c\{Q\})$ treten dann als Nebenbedingungen der Konsequenzregel in Erscheinung.

Die Definition von $\mathbf{vc}(\{P\}c\{Q\})$ erfolgt durch Induktion über den Aufbau von c unter Verwendung einer ebenfalls induktiv definierten Funktion $\mathbf{pre}(c, Q)$, welche die schwächste Vorbedingung liefert, die die Invarianten verwendet.

Definition 3.8 *Seien P, Q Zusicherungen und c ein annotiertes Programm. Wir definieren eine Zusicherung $\mathbf{pre}(c, Q)$ und eine Menge von Aussagen $\mathbf{vc}(\{P\}c\{Q\})$ durch Induktion über den Aufbau von c.*

$$
\begin{array}{lcl}
\mathbf{pre}(\mathbf{skip}, Q) & = & Q \\
\mathbf{vc}(\{P\}\mathbf{skip}\{Q\}) & = & \{P \subseteq Q\} \\
\\
\mathbf{pre}(c_1; c_2, Q) & = & \mathbf{pre}(c_1, \mathbf{pre}(c_2, Q)) \\
\mathbf{vc}(\{P\}c_1; c_2\{Q\}) & = & \mathbf{vc}(\{P\}c_1\{\mathbf{pre}(c_2, Q)\}) \cup \mathbf{vc}(\{\mathbf{pre}(c_2, Q)\}c_2\{Q\}) \\
\\
\mathbf{pre}(X{:=}a, Q) & = & Q[X \mapsto a] \\
\mathbf{vc}(\{P\}X{:=}a\{Q\}) & = & \{P \subseteq Q[X \mapsto a]\} \\
\\
\mathbf{pre}(\mathbf{if}\, b\, \mathbf{then}\, c_1\, \mathbf{else}\, c_2, Q) & = & (b \Rightarrow \mathbf{pre}(c_1, Q)) \wedge (\neg b \Rightarrow \mathbf{pre}(c_2, Q)) \\
\mathbf{vc}(\{P\}\mathbf{if}\, b\, \mathbf{then}\, c_1\, \mathbf{else}\, c_2\{Q\}) & = & \mathbf{vc}(\{P \wedge b\}c_1\{Q\}) \cup \mathbf{vc}(\{P \wedge \neg b\}c_2\{Q\}) \\
\\
\mathbf{pre}(\mathbf{while}_{\{S\}}\, b\, \mathbf{do}\, c, Q) & = & S \\
\mathbf{vc}(\{P\}\mathbf{while}_{\{S\}}\, b\, \mathbf{do}\, c\{Q\}) & = & \{P \subseteq S, S \wedge \neg b \subseteq Q\} \cup \mathbf{vc}(\{S \wedge b\}c\{S\})
\end{array}
$$

Satz 3.9 *Sei c annotiertes Programm und P, Q Zusicherungen. Falls alle Aussagen in $\mathbf{vc}(\{P\}c\{Q\})$ wahr sind, so gilt $\models_p \{P\}c\{Q\}$.*

Zum Beweis zeigen wir durch Induktion über den Aufbau von c, dass $\vdash \{\mathbf{pre}(c, Q)\}c\{Q\}$ und $\vdash \{P\}c\{Q\}$ gilt. Die Aussage folgt dann mit der Korrektheit des Hoare-Kalküls. Die Details verbleiben als Übung. □

Auch wenn man nicht wirklich automatisch Beweise im Hoare-Kalkül erzeugen möchte, ist es eine gute Idee, Programmschleifen per Kommentar mit Invarianten zu annotieren. Die Programmiersprache Eiffel stellt dazu einen halbformalen Rahmen zur Verfügung. Ob es eine Implementierung der **vc**-Funktion für Eiffel gibt, weiß ich nicht.

3.6 Eine verbesserte Konsequenzregel

Wir haben gesehen, dass der Hoare-Kalkül vollständig ist, aber das bezieht sich nur auf die Herleitung von einzelnen Hoare-Tripeln. Anders sieht es aus, wenn wir aus bereits hergeleiteten Hoare-Tripeln weitere gewinnen möchten. Stellen wir uns zum Beispiel vor, wir hätten ein kompliziertes Programm c_{prim} geschrieben, welches testet, ob die über Variable `X` eingegebene Zahl prim ist und das über die Variable `Y` ausgibt. Wir hätten dann mit einigem Aufwand gezeigt, dass

$$\vdash \{\mathtt{X} = x\}c_{\text{prim}}\{x \text{ prim} \Leftrightarrow \mathtt{Y} = 1\}$$

Wenn wir jetzt zum Beispiel $\vdash \{\mathbf{true}\}\mathtt{X}{:=}103; c_{\mathrm{prim}}\{\mathtt{Y} = 1\}$ herleiten möchten, so bleibt uns nichts anderes übrig, als $\vdash \{\mathtt{X} = 103\}c_{\mathrm{prim}}\{\mathtt{Y} = 1\}$ zu zeigen. Wollen wir nicht das Programm c_{prim} auseinandernehmen, kann das nur mit der Konsequenzregel geschehen; die hilft aber auch nicht, weil eben $\mathtt{X} = 103$ nicht stärker ist als $\mathtt{X} = x$.

Abhilfe könnte hier die folgende Substitutionsregel schaffen

$$(\text{H-S{\scriptsize UBST}}) \quad \frac{\{P\}c\{Q\}}{\{P[x \mapsto a]\}c\{Q[x \mapsto a]\}}$$

Diese Regel hat den Nachteil, dass man nur syntaktisch definierbare Substitutionen durchführen kann (hier ist $a \in \mathbf{aexp}$). Was schlimmer ist; die Regel reicht immer nocht nicht aus. Hier ist ein etwas künstliches Beispiel von Morris. Sei c ein Programm mit der Eigenschaft $\{x = 0 \vee x = 1\}c\{\mathtt{X} \neq z\}$. Es gilt dann semantisch auch $\{\mathbf{true}\}c\{\mathtt{X} \neq 0 \wedge \mathtt{Y} \neq 1\}$, denn wenn $\mathtt{X} = 0$ nach Ablauf von c, dann wird die Annahme mit $\pi(x) = 0$ verletzt; ebenso wenn $\mathtt{X} = 1$. Diese Folgerung lässt sich aber nicht einmal mit der Substitutionsregel beweisen, da wir uns dann festlegen müssten, was wir für x einsetzen.

Die endgültige Lösung ist folgende von Schreiber (TAPSOFT '97) vorgeschlagene verbesserte Konsequenzregel:

$$(\text{H-C{\scriptsize ONS}++}) \quad \frac{\{P'\}c\{Q'\} \qquad \Psi}{\{P\}c\{Q\}}$$

wobei Ψ die Nebenbedingung

$$\forall \sigma, \tau \in \Sigma. \forall \pi \in \Pi. P(\sigma, \pi) \Rightarrow Q(\tau, \pi) \vee \exists \pi' \in \Pi. P'(\sigma, \pi') \wedge (Q'(\tau, \pi') \Rightarrow Q(\tau, \pi))$$

ist. Wie ist das zu verstehen? Zunächst stellen wir fest, dass wir die Nebenbedingung der alten Konsequenzregel ja auch so schreiben können

$$\forall \sigma, \tau \in \Sigma. \forall \pi \in \Pi. (P(\sigma, \pi) \Rightarrow P'(\sigma, \pi)) \wedge (Q'(\tau, \pi) \Rightarrow Q(\tau, \pi))$$

Nun ist schon mal klar, dass wir die Implikation $Q'(\tau, \pi) \Rightarrow Q(\tau, \pi)$ nicht brauchen, wenn $P(\sigma, \pi)$ nicht gilt, also können wir auch nehmen

$$\forall \sigma, \tau \in \Sigma. \forall \pi \in \Pi. P(\sigma, \pi) \Rightarrow (P'(\sigma, \pi)) \wedge (Q'(\tau, \pi) \Rightarrow Q(\tau, \pi))$$

Ebenso brauchen wir überhaupt nichts über P', Q' zu sagen, wenn $Q(\tau, \pi)$ sowieso gilt, weswegen wir auch

$$\forall \sigma, \tau \in \Sigma. \forall \pi \in \Pi. P(\sigma, \pi) \Rightarrow Q(\tau, \pi) \vee (P'(\sigma, \pi)) \wedge (Q'(\tau, \pi) \Rightarrow Q(\tau, \pi))$$

nehmen können. Das sieht jetzt schon sehr nach der Nebenbedingung von H-CONS++ aus; der entscheidende Unterschied ist, dass wir den Parameter in P', Q' in Abhängigkeit von σ, τ, π wählen dürfen, also jedesmal etwas anderes einsetzen; z.B. in Morris' Beispiel entweder $x = 0$ oder $x = 1$.

Satz 3.10 *Die Regel* H-CONS++ *ist in folgendem Sinne korrekt. Seien P, P', Q, Q' Zusicherungen und c ein Programm. Falls*

$$\models_p \{P'\}c\{Q'\} \tag{1}$$
$$\forall \sigma, \tau \in \Sigma. \forall \pi \in \Pi. P(\sigma, \pi) \Rightarrow Q(\tau, \pi) \vee \exists \pi' \in \Pi. P'(\sigma, \pi') \wedge (Q'(\tau, \pi') \Rightarrow Q(\tau, \pi)) \tag{2}$$

so folgt $\models_p \{P\}c\{Q\}$.

Zum Beweis nehmen wir an $\sigma, \tau \in \Sigma$, $P(\sigma, \pi)$ und $\sigma \xrightarrow{c} \tau$. Wir haben zu zeigen $Q(\tau, \pi)$. Dazu verwenden wir zunächst (2). Damit haben wir entweder $Q(\tau, \pi)$ direkt und wir sind fertig, oder wir kriegen ein $\pi' \in \Pi$ mit $P'(\sigma, \pi')$ und $Q'(\tau, \pi') \Rightarrow Q(\tau, \pi)$. Mit diesem π' instantiieren wir dann (1) und erhalten $Q'(\tau, \pi')$, somit $Q(\tau, \pi)$ wie verlangt. □

Die Regel H-CONS++ ist in einem gewissen Sinne sogar die bestmögliche. Um dies formulieren zu können, müssen wir den Begriff der partiellen Korrektheit auf beliebige partielle Funktionen erweitern. Sei also $F \subseteq \Sigma \times \Sigma$ eine partielle Funktion. Wir schreiben $\models_p \{P\}F\{Q\}$ für $\forall \sigma \in \Sigma. \forall \pi \in \Pi. P(\sigma, \pi) \Rightarrow \forall \tau \in \Sigma. F(\sigma, \tau) \Rightarrow Q(\tau, \pi)$.

Satz 3.11 *Seien P, P', Q, Q' Zusicherungen. Falls für jede partielle Funktion $F \subseteq \Sigma \times \Sigma$ die Implikation $\models_p \{P'\}F\{Q'\} \Rightarrow \models_p \{P\}F\{Q\}$ gilt, dann ist die Nebenbedingung von* H-CONS++ *erfüllt, also*

$$\forall \sigma, \tau \in \Sigma. \forall \pi \in \Pi. P(\sigma, \pi) \Rightarrow Q(\tau, pi) \vee \exists \pi' \in \Pi. P'(\sigma, \pi') \wedge (Q'(\tau, \pi') \Rightarrow Q(\tau, \pi))$$

Beweis. Seien $\sigma, \tau \in \Sigma$ und $\pi \in \Pi$ mit $P(\sigma, \pi)$ gegeben. Gilt $Q(\tau, \pi)$ so sind wir fertig; wir dürfen also $\neg Q(\tau, \pi)$ voraussetzen. Wir betrachten nun die durch $F(\sigma', \tau') \Leftrightarrow \sigma' = \sigma \wedge \tau' = \tau$ definierte partielle Funktion (F schickt σ auf τ und ist überall sonst undefiniert.)

Nun kann $\models_p \{P'\}F\{Q'\}$ nicht gelten, da sonst nach Annahme auch $\models_p \{P\}F\{Q\}$ und somit $Q(\tau, \pi)$ erfüllt wäre, was wir ja bereits ausgeschlossen haben. Also muss es σ', π', τ' geben mit $P'(\sigma', \pi')$ und $F(\sigma', \tau')$ und $\neg Q'(\tau', \pi')$. Wegen der speziellen Wahl von F liefert das $P'(\sigma, \pi')$ und $\neg Q(\tau, \pi')$, also $P'(\sigma, \pi')$ und $Q(\tau, \pi') \Rightarrow Q(\tau, \pi)$. □

Bemerkung 3.12 Die Regel H-CONS++ mit der stärkeren Nebenbedingung

$$\forall \sigma, \tau \in \Sigma. \forall \pi \in \Pi. P(\sigma, \pi) \Rightarrow \exists \pi' \in \Pi. P'(\sigma, \pi') \wedge (Q'(\tau, \pi') \Rightarrow Q(\tau, \pi))$$

ist auch für totale Korrektheit gültig; das ist aber etwas schwerer zu sehen; Beweis als Übung. Ob in diesem Falle eine zu Satz 3.11 analoge Eigenschaft gilt, weiß ich nicht.

Ihren eigentlichen Nutzen entfaltet die verbesserte Konsequenzregel in Gegenwart rekursiver Prozeduren, die wir als nächstes behandeln wollen.

3.7 Rekursive Prozeduren

Wir betrachten Programme, welche eine einzige parameterlose Prozedur enthalten dürfen. Dazu erweitern wir die Syntax von Befehlen wie folgt um einen Prozeduraufruf:

$$c ::= \dots \mid \mathbf{call} \mid \dots$$

Ein Programm besteht dann aus zwei Befehlen c_0 und c; hierbei ist c_0 der Prozedurrumpf, der selber wieder einen Aufruf enthalten kann und c das eigentliche Programm. Wir wollen es vermeiden, vollständigen Programmen einen eigenen syntaktischen Status zu geben und definieren deshalb die operationale Semantik unter impliziter Bezugnahme auf einen ein für alle mal vorgegebenen Prozedurrumpf c_0. Die einzige neu erforderliche Regel erklärt den neu hinzugekommenen Befehl **call**.

$$(\text{E-C{\small ALL}}) \quad \frac{\sigma \xrightarrow{c_0} \tau}{\sigma \xrightarrow{\mathbf{call}} \tau}$$

Der Aufruf wird also unmittelbar durch den Prozedurrumpf ersetzt. Man könnte meinen, dass parameterlose Prozeduren nutzlos sind; das ist nicht so, hier ist eine Prozedur zur rekursiven Berechnung der Fakultät:

$$\begin{aligned} c_0 \equiv\ & \mathbf{if}\ \mathtt{X}{=}0\ \mathbf{then}\ \mathtt{F}{:=}1\ \mathbf{else} \\ & \quad \mathtt{X}{:=}\mathtt{X}{-}1;\ \mathbf{call}\,;\ \mathtt{X}{:=}\mathtt{X}{+}1; \\ & \quad \mathtt{F}{:=}\mathtt{F}\cdot\mathtt{X} \end{aligned}$$

Es gilt jetzt $\models_p \{\mathtt{X}{=}x\}\,\mathbf{call}\,\{\mathtt{F}{=}x!\}$. Unser Ziel ist es, den Hoare-Kalkül so zu erweitern, dass wir das auch herleiten können.

Die Idee ist, dass $\{P\}\,\mathbf{call}\,\{Q\}$ gilt, wenn es gelingt, $\{P\}c_0\{Q\}$ unter der Zusatzannahme $\{P\}\,\mathbf{call}\,\{Q\}$ zu zeigen. Um das zu formalisieren, müssen wir unseren Hoare-Kalkül um Annahmen erweitern.

Definition 3.13 *Eine* Hoare-Sequenz *ist ein Ausdruck der Form* $\Phi \Longrightarrow \{P\}c\{Q\}$*, wobei* Φ *eine endliche Menge von Hoare-Tripeln der Form* $\{-\}\,\mathbf{call}\,\{-\}$ *ist. Die Menge* Φ *heißt* Annahme*, das Tripel* $\{P\}c\{Q\}$ *heißt Konklusion der Sequenz.*

Die bisherigen Regeln erweitern sich auf Hoare-Sequenzen, indem man die Annahmen einfach mitführt. Der Vollständigkeit halber sind sie in Figur 1 zusammengefasst. Darin sind auch schon eine Formalisierung des rekursiven Beweisprinzips für Prozeduren, sowie ein Axiom enthalten, welches gestattet, Annahmen aus Φ tatsächlich zu verwenden. Wir schreiben, $\vdash \Phi \Longrightarrow \{P'\}c\{Q'\}$, wenn diese Sequenz mit Hilfe der Regeln aus Figur 1 hergeleitet werden kann.

$$(\textsc{H-Skip}) \quad \frac{}{\Phi \Longrightarrow \{P\}\mathbf{skip}\{P\}}$$

$$(\textsc{H-Seq}) \quad \frac{\Phi \Longrightarrow \{P\}c_1\{P'\} \quad \Phi \Longrightarrow \{P'\}c_2\{Q\}}{\Phi \Longrightarrow \{P\}c_1; c_2\{Q\}}$$

$$(\textsc{H-Ass}) \quad \frac{}{\Phi \Longrightarrow \{P[X \mapsto a]\}X{:=}a\{P\}}$$

$$(\textsc{H-If}) \quad \frac{\Phi \Longrightarrow \{P \wedge b\}c_1\{Q\} \quad \Phi \Longrightarrow \{P \wedge \neg b\}c_2\{Q\}}{\Phi \Longrightarrow \{P\}\mathbf{if}\, b\, \mathbf{then}\, c_1\, \mathbf{else}\, c_2\{Q\}}$$

$$(\textsc{H-While}) \quad \frac{\Phi \Longrightarrow \{P \wedge b\}c\{P\}}{\Phi \Longrightarrow \{P\}\mathbf{while}\, b\, \mathbf{do}\, c\{P \wedge \neg b\}}$$

$$(\textsc{H-Ax}) \quad \frac{\{P\}\,\mathbf{call}\,\{Q\} \in \Phi}{\Phi \Longrightarrow \{P\}\,\mathbf{call}\,\{Q\}}$$

$$(\textsc{H-Cons++}) \quad \frac{\Phi \Longrightarrow \{P'\}c\{Q'\}}{\Phi \Longrightarrow \{P\}c\{Q\}}$$

wenn $\forall \sigma, \tau. \forall \pi. P(\sigma, \pi) \Rightarrow Q(\tau, \pi) \vee \exists \pi'. P'(\sigma, \pi') \wedge (Q'(\tau, \pi') \Rightarrow Q(\tau, \pi))$

$$(\textsc{H-Proc}) \quad \frac{\Phi \cup \{\{P\}\,\mathbf{call}\,\{Q\}\} \Longrightarrow \{P\}c_0\{Q\}}{\Phi \Longrightarrow \{P\}\,\mathbf{call}\,\{Q\}}$$

Abbildung 1: Hoare-Regeln mit Annahmen

3.7.1 Beispiel

Betrachten wir obige Prozedur c_0 zur Berechnung der Fakultät. Hier ist eine Herleitung von $\vdash\ \emptyset \Longrightarrow \{\mathtt{X}=x \land x \geq 0\}\,\mathbf{call}\,\{\mathtt{F}=x!\}$. Die Herleitung sollte man von unten nach oben lesen, wodurch sich auch die Bedeutung der eingeführten Abkürzungen ergibt.

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\cdots \qquad
\dfrac{
\dfrac{
\dfrac{
\cdots \qquad
\dfrac{
\dfrac{}{\vdash \Phi \Longrightarrow \{\mathtt{X}=x \land x \geq 0\}\,\mathbf{call}\,\{\mathtt{X}=x \land \mathtt{F}=x!\}}\ \text{H-Ax}
}{\vdash \Phi \Longrightarrow \{\mathtt{X}=x-1 \land x > 0\}\,\mathbf{call}\,\{\mathtt{X}=x-1 \land \mathtt{F}\cdot x=x!\}}\ \text{H-Cons++}
}{\vdash \Phi \Longrightarrow \{\mathtt{X}=x \land x > 0\}\mathtt{X}{:=}\mathtt{X}-1;\ \mathbf{call}\,\{\mathtt{X}=x-1 \land \mathtt{F}\cdot x=x!\}}\ \text{H-Seq}
}{\vdash \Phi \Longrightarrow \{\mathtt{X}=x \land x > 0\}c\{\mathtt{X}=x \land \mathtt{F}\cdot\mathtt{X}=x!\}}\ \text{H-Ass+H-Seq}
}{\vdash \Phi \Longrightarrow \{\mathtt{X}=x \land x > 0\}c;\mathtt{F}{:=}\mathtt{F}\cdot\mathtt{X}\{\mathtt{X}=x \land \mathtt{F}=x!\}}\ \text{H-Ass+H-Seq}
}{\vdash \{\{\mathtt{X}=x \land x \geq 0\}\,\mathbf{call}\,\{\mathtt{X}=x \land \mathtt{F}=x!\}\} \Longrightarrow \{\mathtt{X}=x \land x \geq 0\}c_0\{\mathtt{X}=x \land \mathtt{F}=x!\}}\ \text{H-If}
}{\vdash \emptyset \Longrightarrow \{\mathtt{X}=x \land x \geq 0\}\,\mathbf{call}\,\{\mathtt{X}=x \land \mathtt{F}=x!\}}\ \text{H-Proc}
}{\vdash \emptyset \Longrightarrow \{\mathtt{X}=x \land x \geq 0\}\,\mathbf{call}\,\{\mathtt{F}=x!\}}\ \text{H-Cons}
$$

Man beachte die entscheidende Verwendung der verbesserten Konsequenzregel im obersten Schritt. Wir haben hier verwendet, dass $x! = x \cdot (x-1)!$, wenn $x > 0$. Krzysztof Apt zeigt in (10 years of Hoare Logic, a survey. ACM Transactions on Programming Languages and Syntax, 3(4), 1981), dass man mit der normalen Konsequenzregel dieses Programm nicht korrekt beweisen kann.

3.7.2 Korrektheit des Hoare-Kalküls für rekursive Prozeduren

Im folgenden werden wir die Metavariablen ϕ, ψ für Hoare Tripel einführen, sodass eine Sequenz die Form $\Phi \Longrightarrow \phi$ hat. Weiter führen wir die Notation $\models_p \Phi$ für $\bigwedge_{\phi \in \Phi} \models_p \phi$ ein.

Wir wollen nun zeigen, dass dieser erweiterte Kalkül korrekt ist, also dass aus $\vdash \emptyset \Longrightarrow \phi$ folgt $\models_p \phi$. Damit dieser Beweis durchgeht, werden wir auch irgendeine Korrektheitsaussage über Sequenzen mit Annahmen machen müssen.

Zu diesem Zweck brauchen wir endliche Approximationen für Prozeduren.

Definition 3.14 *Seien c, d Programme. Das Programm $c[d/\,\mathbf{call}\,]$ entsteht aus c durch Ersetzung aller Vorkommen von* **call** *durch d. Formal haben wir*

$$
\begin{array}{lcl}
\mathbf{skip}[d/\,\mathbf{call}\,] & = & \mathbf{skip} \\
(X{:=}a)[d/\,\mathbf{call}\,] & = & X{:=}a \\
(c_1; c_2)[d/\,\mathbf{call}\,] & = & c_1[d/\,\mathbf{call}\,]; c_2[d/\,\mathbf{call}\,] \\
(\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2)[d/\,\mathbf{call}\,] & = & \mathbf{if}\ b\ \mathbf{then}\ c_1[d/\,\mathbf{call}\,]\ \mathbf{else}\ c_2[d/\,\mathbf{call}\,] \\
(\mathbf{while}\ b\ \mathbf{do}\ c)[d/\,\mathbf{call}\,] & = & \mathbf{while}\ b\ \mathbf{do}\ c[d/\,\mathbf{call}\,] \\
\mathbf{call}\,[d/\,\mathbf{call}\,] & = & d
\end{array}
$$

Diese Substitution setzen wir auf Hoare-Tripel fort durch $(\{P\}c\{Q\})[d/\,\mathbf{call}\,] = \{P\}c[d/\,\mathbf{call}\,]\{Q\}$.

Definition 3.15 *Für $n \in \mathbf{N}$ definieren wir $\mathbf{call}^{(n)} \in \mathbf{com}$ rekursiv durch*

$$\begin{aligned}\mathbf{call}^{(n)} &\stackrel{\text{def}}{=} \mathbf{diverge}\\ \mathbf{call}^{(n+1)} &\stackrel{\text{def}}{=} c_0[\,\mathbf{call}^{(n)}/\,\mathbf{call}\,]\end{aligned}$$

wobei **diverge** *ein nichtterminierendes Programm ist, z.B.* **while true do skip**.

Folgendes Lemma ist eine intuitiv klare Folgerung aus den Definitionen der operationalen Semantik und der partiellen Korrektheit. Formal zeigt man die ersten beiden Teile durch Induktion über die operationale Semantik; die letzten beiden Teile sind rein logische Konsequenzen aus den ersten beiden.

Lemma 3.16 *Seien σ, τ Zustände und c Programm.*

1. *Wenn $\sigma \xrightarrow{c[\mathbf{call}^{(n)}/\,\mathbf{call}\,]} \tau$ für ein $n \in \mathbf{N}$, dann auch $\sigma \xrightarrow{c[\mathbf{call}^{(m)}/\,\mathbf{call}\,]} \tau$ für alle $m \geq n$.*

2. *$\sigma \xrightarrow{c} \tau$ genau dann, wenn $\exists n \in \mathbf{N}. \sigma \xrightarrow{c[\mathbf{call}^{(n)}/\,\mathbf{call}\,]} \tau$*

3. *Wenn $\models_p \phi[\,\mathbf{call}^{(n)}/\,\mathbf{call}\,]$ für ein $n \in \mathbf{N}$, dann auch $\models_p \phi[\,\mathbf{call}^{(m)}/\,\mathbf{call}\,]$ für alle $m \leq n$.*

4. *$\models_p \phi$ genau dann, wenn $\forall n \in \mathbf{N}. \phi[\,\mathbf{call}^{(n)}/\,\mathbf{call}\,]$.*

Ausgerüstet mit diesen Eigenschaften können wir nun den Korrektheitssatz formulieren:

Satz 3.17 *Sei $\Phi \Longrightarrow \phi$ eine Hoare-Sequenz. Wenn $\vdash \Phi \Longrightarrow \phi$, dann gilt für alle $n \in \mathbf{N}$ die Implikation*

$$\models_p \Phi[\,\mathbf{call}^{(n)}/\,\mathbf{call}\,] \;\Rightarrow\; \models_p \phi[\,\mathbf{call}^{(n)}/\,\mathbf{call}\,]$$

Der Beweis erfolgt wieder durch Induktion über Herleitungen. Die Fälle für die alten Regeln gehen wir gehabt; die Regel H-Ax ist trivial; es bleibt Regel H-Proc. Nehmen wir also an $\vdash \Phi \cup \{\{P\}\,\mathbf{call}\,\{Q\}\} \Longrightarrow \{P\}c_0\{Q\}$. Die Induktionshypothese liefert für alle $n \in \mathbf{N}$

$$\models_p \Phi[\,\mathbf{call}^{(n)}/\,\mathbf{call}\,] \wedge \models_p \{P\}\,\mathbf{call}^{(n)}\{Q\} \Rightarrow \models_p \{P\}\,\mathbf{call}^{(n+1)}\{Q\}$$

wobei wir mit Hilfe der Definition der $\mathbf{call}^{(n)}$ etwas vereinfacht haben. Wir müssen für alle $n \in \mathbf{N}$ zeigen

$$A(n) \quad \stackrel{\text{def}}{\Leftrightarrow} \quad \models_p \Phi[\,\mathbf{call}^{(n)}/\,\mathbf{call}\,] \Rightarrow \models_p \{P\}\,\mathbf{call}^{(n)}\{Q\}$$

Das tun wir durch Induktion über n. Der Induktionsanfang ist einfach, da $\mathbf{call}^{(0)} = \mathbf{diverge}$ und $\models_p \{P\}\mathbf{diverge}\{Q\}$ stets gilt. Für den Induktionsschritt nehmen wir $A(n)$ an und weiter $\models_p \Phi[\mathbf{call}^{(n+1)}/\mathbf{call}]$. Wir müssen zeigen $\models_p \{P\}\,\mathbf{call}^{(n+1)}\{Q\}$. Mit Lemma 3.16(3) erhalten wir $\models_p \Phi[\mathbf{call}^{(n)}/\mathbf{call}]$ und mit $A(n)$ dann auch $\models_p \{P\}\,\mathbf{call}^{(n)}\{Q\}$. Beides zusammen liefert mit der globalen Induktionshypothese die gewünschte Aussage. $\Box$

Man beachte, dass aus dem Korrektheitssatz wegen Lemma 3.16(4) insbesondere folgt, dass wenn $\emptyset \Longrightarrow \{P\}c\{Q\}$ herleitbar ist, dann $\models_p \{P\}c\{Q\}$.

3.7.3 Vollständigkeit des Hoare-Kalküls für rekursive Prozeduren

Bei dem Beweis der Vollständigkeit stoßen wir auf das Problem, dass wir beim Beweis von $\{\mathbf{wp}(c,Q)\}c\{Q\}$ auf $\{\mathbf{wp}(\,\mathbf{call}\,,Q')\}\,\mathbf{call}\,\{Q'\}$ mit $Q' \neq Q$ zurückgreifen müssten. Das haben wir aber bei einem induktiven Beweis nicht zur Verfügung. Die Lösung ist es, $\{\mathbf{wp}(c,Q)\}c\{Q\}$ nur für eine feste, sozusagen allgemeinste, Zusicherung Q_0 zu zeigen. Damit wir diese formulieren können, muss der Parameterraum für jede Programmvariable eine Hilfsvariable bereitstellen; das ist etwa dann der Fall, wenn $\Pi = \Sigma$. Davon wollen wir im folgenden ausgehen. Die allgemeinste Zusicherung Q_0 hat dann folgende denkwürdige Form

$$Q_0(\sigma,\pi) \overset{\text{def}}{\Leftrightarrow} \sigma{=}\pi$$

Mit P^c bezeichnen wir die schwächste Vorbedingung für Q_0 unter c. Es ist

$$P^c(\sigma,\pi) \overset{\text{def}}{\Leftrightarrow} \forall\tau.\sigma \xrightarrow{c} \tau \Rightarrow \tau{=}\pi$$

Da die Auswertung deterministisch ist, bedeutet $P^c(\sigma,\pi)$, dass c in σ entweder nicht terminiert, oder Endzustand π hat.

Satz 3.18 *Wenn $\models_p \{P\}c\{Q\}$, dann gilt*

$$\forall\sigma,\tau\in\Sigma.\forall\pi\in\Pi.P(\sigma,\pi) \Rightarrow Q(\tau,\pi) \vee \exists\pi'\in\Pi.P^c(\sigma,\pi') \wedge (Q_0(\tau,\pi') \Rightarrow Q(\tau,\pi))$$

also lässt sich $\vdash \{P\}c\{Q\}$ aus $\vdash \{P^c\}c\{Q_0\}$ mit Hilfe von H-CONS++ *gewinnen.*

Beweis als Übung. $\Box$

Satz 3.19 *Für jedes Programm c gilt*

- $\vdash \{\{P^{\mathbf{call}}\}\,\mathbf{call}\,\{Q_0\}\} \Longrightarrow \{P^c\}c\{Q_0\}$
- $\vdash \emptyset \Longrightarrow \{P^c\}c\{Q_0\}$

Den ersten Teil zeigen wir durch Induktion über c. Der Fall $c =$ **call** ist eine Instanz von H-Ax. Betrachten wir den Fall $c = \textbf{while}\, b\, \textbf{do}\, c_1$. Folgende Herleitung liefert das Resultat zusammen mit der Induktionshypothese:

$$\cfrac{\cfrac{\cfrac{\cfrac{}{\{\{P^{\mathbf{call}}\}\,\mathbf{call}\,\{Q_0\}\}\Longrightarrow\{P^{c_1}\}c_1\{Q_0\}}\ \textsc{Ind-Hyp}}{\{\{P^{\mathbf{call}}\}\,\mathbf{call}\,\{Q_0\}\}\Longrightarrow\{P^{c}\wedge b\}c_1\{P^{c}\}}\ \text{H-}\textsc{Cons++}}{\{\{P^{\mathbf{call}}\}\,\mathbf{call}\,\{Q_0\}\}\Longrightarrow\{P^{c}\}c\{P^{c}\wedge\neg b\}}\ \text{H-}\textsc{While}}{\{\{P^{\mathbf{call}}\}\,\mathbf{call}\,\{Q_0\}\}\Longrightarrow\{P^{c}\}c\{Q_0\}}\ \text{H-}\textsc{Cons}$$

Die Bestätigung der Nebenbedingungen und die verbleibenden Fälle lassen wir als Übung.

Den zweiten Teil zeigen wir genau wie den ersten, nur dass wir hier im Falle von $c =$ **call** die Regel H-Proc und den schon bewiesenen ersten Teil (für $c = c_0$) verwenden. □

Zusammen mit Satz 3.18 erhalten wir jetzt die eigentliche Korrektheit:

Korollar 3.20 *Wenn* $\models_p \{P\}c\{Q\}$, *dann* $\vdash \emptyset\Longrightarrow\{P\}c\{Q\}$.

3.7.4 Totale Korrektheit

Einen korrekten und vollständigen Kalkül für die totale Korrektheit erhält man, indem man die Regeln H-While und H-Proc durch folgende ersetzt

$$(\text{H-}\textsc{While-Tot})\quad \frac{\Phi\Longrightarrow\{P(n+1)\}c\{P(n)\}\quad \forall n.P(n+1)\Rightarrow b\wedge P(0)\Rightarrow\neg b}{\Phi\Longrightarrow\{\exists n.P(n)\}\textbf{while}\, b\, \textbf{do}\, c\{P(0)\}}$$

$$(\text{H-}\textsc{Proc-Tot})\quad \frac{\neg P(0)\qquad \Phi\cup\{\{P(n)\}\,\mathbf{call}\,\{Q\}\}\Longrightarrow\{P(n+1)\}c_0\{Q\}}{\Phi\Longleftrightarrow\{\exists n.P(n)\}\,\mathbf{call}\,\{Q\}}$$

Außerdem muss die Konsequenzregel durch die in Bem. 3.12 erwähnte Abschwächung ersetzt werden. Die Variable n in Regel H-Proc ist eine *Zählvariable* und keine Hilfsvariable. Zählvariablen werden über die ganze Sequenz allquantifiziert und dürfen deshalb auch nicht im Rahmen der Konsequenzregel verändert werden. Man muss also die Zählvariablen, sowohl von den Programmvariablen, als auch von den Hilfsvariablen getrennt halten. Formalisiert man dies richtig, dann wird der Beweis der Korrektheit sogar einfacher als bei der partiellen Korrektheit, da man auf die Approximationen verzichten kann und gleich zeigt $\models_t \Phi \Rightarrow \models_t \phi$ falls $\vdash \Phi\Longrightarrow\phi$. Details finden sich in der Arbeit von Schreiber und in (America, de Boer. Proving total correctness of recursive procedures. Information and Computation. 84(2), 1990.).

3.8 Expressivität

Die Vollständigkeitsbeweise, die wir gegeben haben beruhten immer auf der Betrachtung bestimmter Zusicherungen, deren Definition die operationale Semantik beinhaltet, z.B. $\mathbf{wp}(c, Q)$ oder P^c. Zusicherungen waren für uns immer beliebige Mengen, bzw. Relationen, sodass diese speziellen Zusicherungen immer „existieren". Anders sieht es aus, wenn man sich bei der Formulierung der Zusicherungen auf einen bestimmten Formalismus (*Zusicherungssprache*) beschränkt. Dann kommt es darauf an, zu zeigen, dass die Zusicherungen $\mathbf{wp}(c, Q)$ und P^c tatsächlich in dem Formalismus hinschreibbar sind.

Man bezeichnet er eine Zusicherungssprache als *expressiv* bzgl. einer Programmiersprache, falls für jede hinschreibbare Zusicherung Q und Programm c auch $\mathbf{wp}(c, Q)$ hinschreibbar ist.

Es ist unmittelbar klar, dass der Hoare-Kalkül auch dann noch vollständig ist, wenn man sich auf Zusicherungen einschränkt, die in einer expressiven Zusicherungssprache formulierbar sind.

Stephen Cook war der erste, der die Bedeutung dieses Begriffs gesehen hat. In seiner Arbeit [*Soundness and completeness of an axiom system for program verification.* SIAM Journal on Computing, 7(1):70–90, 1978] zeigt er u.a., dass die Peano Arithmetik (Logik erster Stufe über $+$ und $\times$) expressiv für **while**-Programme ist. Der Beweis verwendet eine Kodierung von Zustandsfolgen als natürliche Zahlen. Nicht expressiv ist zum Beispiel die sog. Presburger Arithmetik, bei der man nur die Gleichheit und die Addition, nicht aber die Multiplikation bereitstellt. Der Grund ist, dass Gültigkeit von Formeln in der Presburger Arithmetik entscheidbar ist. Wäre die Presburger Arithmetik jetzt expressiv, dann könnte man ein die Gültigkeit eines Hoare Tripels entscheiden, indem man einfach die Verifikationsbedingungen checkt, die sich aus der Annotation der **while**-Schleifen mit den jeweils schwächsten Vorbedingungen ergeben. Die Frage, ob $\{\mathbf{true}\}c\{\mathbf{false}\}$ gültig ist, ist aber äquivalent zum Halteproblem und damit unentscheidbar.

Die Frage, ob eine bestimmte Zusicherungssprache expressiv bzgl. einer bestimmten Erweiterung (um Daten- oder Kontrollstrukturen) von IMP ist, wurde in der Literatur ausgiebig behandelt. Siehe z.B. die Standardreferenz [Mitchell Wand. *A new incompleteness result for Hoare's logic.* Journal of the ACM 25(1). 1978]

Natürlich reicht es nicht aus, dass man eine expressive Zusicherungssprache verwendet; sie muss natürlich auch gestattet, interessante Spezifikationen zu formulieren. Beachte hier, dass die Zusicherungssprache, die nur die wahre Zusicherung **true** enthält expressiv ist!

3.9 Weiterführendes zur Hoare Logik

In der Literatur finden sich Hoare-artige Beweisregeln für so ziemlich jedes programmiersprachliche Konstrukt im Bereich der imperativen Programmierung.

Leider werden die Regeln mit zunehmender Komplexität der Konstrukte immer komplizierter. Probleme gibt's zum Beispiel mit Termen, die Seiteneffekte haben, etwa der Term $\mathtt{F}()$, wenn

$$
\begin{array}{l}
\textbf{function}\,\mathtt{F}() = \{\\
\quad \mathtt{X} := \mathtt{X} + 1\\
\quad \mathtt{F} := \mathtt{X}\}
\end{array}
$$

Es gilt dann nicht, dass eine Wertzuweisung $X{:=}a$ nur die zugewiesene Variable verändert (betrachte $\mathtt{Y}{:=}\mathtt{F}()$). Somit gilt auch die Regel H-Ass dann nicht mehr.

Ein ähnliches Problem kriegt man mit indizierten Variablen. Da kann es passieren, dass eine Variable nach einer Zuweisung nicht den zugewiesenen Wert enthält. Z.B. gilt nicht $\{\mathbf{true}\}\mathtt{A}[\mathtt{A}[0]] := 1\{\mathtt{A}[\mathtt{A}[0]] = 1\}$. (Betrachte $\mathtt{A}[0] = 0, \mathtt{A}[1] = 1$).

Ein interessanter Gedanke, der schon 1978 von Reynolds aufgebracht wurde [John Reynolds. *Syntactic Control of Interference*, Symp. Principles of Programming Languages, 1978] besteht darin, durch ein geeignetes Typisierungssystem solche Anomalien von vornherein auszuschließen. Reynolds' Ideen wurden erst in allerjüngster Zeit wieder aufgegriffen und es bleibt zu sehen, ob sich daraus eine „einfache" Hoare-Logik für eine reichhaltige Programmiersprache mit Funktionen, Parametern, Arrays etc. gewinnen lässt.

Ein weiteres Gebiet in welchem Hoare-Logik heute verwendet wird, ist die Verifikation verteilter Systeme. Man kann sich so ein System als parallele Komposition mehrerer IMP-Programme vorstellen, die über gemeinsame Programmvariablen kommunizieren. Leider ist es hier so, dass sich partielle Korrektheitsaussagen über die einzelnen Komponenten nicht unbedingt auf das Gesamtsystem übertragen. Herauszufinden, wann sie das tun, oder wie man Aussagen über die Komponenten zu einer Aussage über das Gesamtsystem liften kann, ist Gegenstand intensiver Forschung. Siehe z.B. das Buch von Apt Olderog [Verification of Sequential and Concurrent Programs, Springer 1991] und Skripten von Hooman/de Boer/de Roever [Bei Interesse mich fragen].

Einfacher ist es, wenn man die IMP-Programme als atomar auffasst und ein verteiltes Programm als Menge von mit Ausführbarkeitsbedingungen versehenen IMP-Programmen (sog. Transitionen) auffasst. Der Ablauf besteht dann darin, dass jeweils aus den gerade ausführbaren Transitionen eine ausgewählt wird und zwar so, dass jede auch irgendwann drankommt. Hier kann man jede einzelne Transition mit dem Hoare-Kalkül lokal verifizieren. Details stehen im Buch von Chandy und Misra [Parallel Program Design, Addison Wesley, 1988].

4 Semantik für funktionale Programmierung

In den Übungen haben wir gesehen, dass eine rein imperative Programmiersprache etwas schwerfällig ist, wenn es darum geht, rekursiv definierte Funktionen zu programmieren, wie etwa die Fibonaccizahlen oder die Fakultät. Und wenn man es dann geschafft hat, dann ist die Verifikation vergleichsweise aufwendig, da man sich um das Nichtverändertwerden bestimmter globaler Variablen kümmern muss.

In praktisch relevanten Programmiersprachen wird daher die rekursive Definition von Funktionen als Primitivum bereitgestellt. Man kann also etwa in C die Fibonaccizahlen durch

```
int fib(int x)
{
if (x==0 || x==1)
    return(1);
else
    return(fib(x-1)+fib(x-2));
}
```

definieren. In C lassen sich auch Funktionen höherer Ordnung definieren, beispielsweise ist

```
int applytwice(int f(int),int x)
{
return f(f(x));
}
```

eine Funktion, die ihr erster Argument (`f`) zweimal auf ihr zweites Argument (`x`) anwendet. Der Aufruf `applytwice(fib,4)` liefert also `fib(5)=8` zurück.

Da nun das Zusammenspiel von (evtl. rekursiven) Funktionen höherer Ordnung mit der Wertzuweisung vom Gesichtspunkt der Semantik und Verifikation her gesehen ziemlich schwierig zu beschreiben ist, bietet es sich an, zuerst mal (und für unsere Vorlesung heißt das *nur*) eine rein funktionale Sprache zu betrachten, in der es nur rekursive Funktionen und keinerlei Wertzuweisung gibt. Dass das geht, sieht man obigem Beispiel, welches ja ohne Wertzuweisung auskommt.

Realen funktionalen Programmiersprachen wie ML und Haskell liegt die Idee zugrunde, dass imperatives Programmieren (mit Funktionen) nicht nur zu theoretischen Schwierigkeiten führt, sondern auch in den praktischen Anwendungen zu schwierig zu findenden Fehlern führt. Es ist sicher richtig, dass man rein funktionale Programme einfacher verifizieren kann, als imperative Programme. Gegner der funktionalen Programmierung führen allerdings ins Feld, dass funktionale Programme ineffizient sind und dass man in richtigen Anwendungen ohne Wertzuweisung und Sprungbefehle nicht auskommt. In der Tat stellen reale funktionale Programmiersprachen solche imperativen Konstrukte zur Verfügung und diese werden auch tatsächlich in jedem größeren Programm verwendet.

Leider steckt aber die Semantik und Verifikation funktionaler Programme mit imperativen Konstrukten noch ziemlich in den Kinderschuhen, sodass zumindest im Moment nicht so ganz klar ist, ob funktionale Programme vom Gesichtspunkt der Zuverlässigkeit her wirklich einen Vorteil bieten. Es hat allerdings in letzter Zeit große Fortschritte gegeben, sowohl auf dem Gebiet der Effizienz, als auch der Semantik und Verifikation realer funktionaler Sprachen, sodass mittelfristig wohl doch ein Vorteil zu erwarten ist.

Das Für und Wider funktionaler Programmierung ist aber nicht Gegenstand dieser Veranstaltung; dazu gibt es andere Vorlesungen und Bücher, z.B. [Larry Paulson: ML for the working programmer, Cambridge University Press, 1991]. Hier wollen wir die Semantik einer einfachen „theoretischen" funktionalen Programmiersprache, nämlich Gordon Plotkin's PCF *programming language for computable functions*, kennenlernen. Die dabei verwendeten Methoden kommen allerdings in verfeinerter Form auch bei theoretischen Untersuchungen über „echte" Programmiersprachen mit funktionalen Konzepten (wie ML, aber auch Algol oder C) zur Anwendung.

4.1 Die Programmiersprache PCF

In IMP gab es nur zwei Typen von Ausdrücken: Boole'sche und Arithmetische; welchen Typ ein Ausdruck hatte, konnte man schon auf der Ebene der Syntax entscheiden. In einer (typisierten) funktionalen Programmiersprache gibt es zusätzlich noch Funktionstypen: Sind τ_1, τ_2 Typen, so bezeichnet $\tau_1 \to \tau_2$ den Typ der Funktionsausdrücke, die ein Argument des Typs τ_1 erwarten und dann einen Wert vom Typ τ_2 zurückgeben.

Die Typen von PCF sind also durch folgende BNF-Grammatik gegeben:

$$\tau ::= \mathbf{int} \mid \tau_1 \to \tau_2$$

Boole'sche Werte lassen wir der Einfachheit halber weg (0 steht für **true**, alles andere für **false**); **int** beinhaltet im Gegensatz zu IMP hier nur natürliche Zahlen (keine negativen).

Zum Beispiel ist $\mathbf{int} \to \mathbf{int}$ der Typ der Funktionen, die einen *integer* als Argument nehmen und einen ebensolchen zurückliefern.

Längere Ketten von Pfeilen klammert man nach rechts, also bedeutet $\mathbf{int} \to \mathbf{int} \to \mathbf{int}$ den Typ $\mathbf{int} \to (\mathbf{int} \to \mathbf{int})$. Ein Element dieses Typs ist eine Funktion, die auf einen *integer* anzuwenden sind und dann als Resultat eine einstellige Funktion zurückliefern. Im Prinzip ist ein Element f von $\mathbf{int} \to \mathbf{int} \to \mathbf{int}$ also eine zweistellige Funktion. Zum Beispiel hat man $\mathbf{plus} : \mathbf{int} \to \mathbf{int} \to \mathbf{int}$. Es ist dann $\mathbf{plus}\, 3\, 5 = 8 : \mathbf{int}$ und $\mathbf{plus}\, 3 : \mathbf{int} \to \mathbf{int}$ ist die Funktion, die ihr Argument um 3 vermehrt. Definieren wir also $f = \mathbf{plus}\, 3$, dann wird $f\, 4 = 7$.

Von $\mathbf{int} \to \mathbf{int} \to \mathbf{int}$ wohl zu unterscheiden ist der Typ $(\mathbf{int} \to \mathbf{int}) \to \mathbf{int}$. Er enthält Funktionen, die als Argument eine *Funktion* des Typs $\mathbf{int} \to \mathbf{int}$ nehmen und dann einen integer zurückliefern. Solche Funktionen werden in der Mathematik oft als *Funktionale* bezeichnet. Ein Beispiel ist das durch $F_1(f) =$

$f(0)$ definierte Auswertungsfunktional. Interessanter sind Funktionale wie

$$F_2(f) = \text{„das kleinste } x\text{, sodass } f(x) = 0\text{“}$$

Es gibt auch Funktionale, die sowohl Funktionen, als auch normale integer als Argumente nehmen. Z.B.

$$F_3(f)(x) = f(x)$$

oder

$$F_4(f)(x)(y) = \underbrace{f(f(\ldots f(}_{x \text{ mal}}(y)\ldots)$$

Es ist $F_3 : (\mathbf{int} \to \mathbf{int}) \to \mathbf{int} \to \mathbf{int}$ und $F_4 : (\mathbf{int} \to \mathbf{int}) \to \mathbf{int} \to \mathbf{int} \to \mathbf{int}$.

Die PCF Typen erlauben auch Funktionale noch höherer Stufe, also Funktionen, die etwa F_3 als Argument verarbeiten können, usw. In der Praxis kommen aber so komplizierte Typen nur selten vor.

Kommen wir zur konkreten Form der Ausdrücke. Aus technischen Gründen ist es günstig, zunächst durch eine BNF-Grammatik die Menge der syntaktisch richtigen, aber möglicherweise falsch typisierten Ausdrücke festzulegen (wenn etwa $f : \mathbf{int} \to \mathbf{int}$, dann ist $f\ f$ syntaktisch richtig, aber falsch typisiert) und dann die wohltypisierten Terme (Term=Ausdruck) durch eine induktive Definition auszusondern. Diese Grammatik ist wie folgt.

$e ::=$		n	numerische Konstante $\in \mathbb{N}$
	$\mid$	$e_1\, e_2$	Funktionsanwendung
	$\mid$	$\mathbf{fun}(x{:}\,\tau)e$	Funktionsabstraktion
	$\mid$	x	Variable
	$\mid$	$\mathbf{fix}_\tau$	rekursive Definition
	$\mid$	$e_1\ \mathbf{op}\ e_2$, für $\mathbf{op} \in \{+, -, \times\}$	binäre Operationen
	$\mid$	$\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3$	Fallunterscheidung

Jede natürliche Zahl ist also insbesondere ein PCF-Term. Wir könnten natürlich auch für jede natürliche Zahl n eine spezielle Konstante c_n einführen.

Eine Applikation $e_1\ e_2$ bedeutet die Anwendung der durch e_1 beschriebenen Funktion auf e_2. Man beachte, dass man das Argument e_2 nicht einklammern muss. Man kann es aber tun um die Lesbarkeit zu verbessern. In Implementierungen von funktionalen Sprachen gibt es eine ganze Reihe von Vorrang- und Klammerregeln (wie Punkt vor Strich), die es erlauben, einen Ausdruck mit möglichst wenig Klammern zu schreiben und ihn dennoch eindeutig syntaktisch zu analysieren (*parsen*). Im Skript nehmen wir darauf keine Rücksicht und verstehen die konkrete Syntax als Suggestivnotation für eindeutige Syntaxbäume.

Die Funktionsabstraktion $\mathbf{fun}(x{:}\,\tau)e$ erlaubt es uns, anonyme Funktionen hinzuschreiben. In der üblichen mathematischen Sprech- und Schreibweise kann man

Funktionen immer nur zusammen mit einem Namen für sie hinschreiben. Man sagt also etwa: „Sei f die durch $f(x) = \ldots$ definierte Funktion … “. Möchte man also ein Funktional $F : (\mathbf{int} \to \mathbf{int}) \to \mathbf{int}$ auf eine bestimmte Funktion anwenden, sagen wir die Nachfolgerfunktion, dann müsste man so etwas wie „$F(f)$, wobei $f(x) = x + 1$“ schreiben. Mit der Funktionsabstraktion kann man sich das f sparen und einfach schreiben $F(\mathbf{fun}(x{:}\,\mathbf{int})x + 1)$. Die Funktionsabstraktion nimmt man in PCF aber auch, um Funktionen zu definieren, wenn man einen Namen benutzen möchte. Statt $f(x) = x+1$ schreibt man $f = \mathbf{fun}(x{:}\,\mathbf{int})x+1$ und hat damit die Funktionsdefinition auf die ganz normale Definition von Abkürzungen zurückgeführt.

Oft schreibt man auch $\lambda x{:}\,\tau.e$ für Funktionsabstraktionen. Wir behalten uns das λ für Metalevel-Abstraktionen vor, also zum Beispiel können wir $\lambda x{\in}\mathbf{N}.x+1$ für die „wirkliche“ Nachfolgerfunktion auf den natürlichen Zahlen schreiben.

Im Rumpf einer Funktionsabstraktion kann und soll nun die gebundene Variable vorkommen. Daher muss man auch Terme mit Variablen und insbesondere die Variablen selber als Terme zulassen. Dazu dient die vierte Klausel in der Grammatik. Hier rangiert x wie in IMP über eine ein für allemal festgelegte unendliche Menge, etwa Zeichenketten bestimmter Form. Wie schon eingangs erwähnt handelt es sich bei den Variablen in PCF *nicht* um durch Wertzuweisung veränderbare imperative Variablen, sondern immer um formale Parameter einer Funktionsdefinition. Der Term x ist für sich genommen also sinnlos; nur als Teilterm einer Funktionsabstraktion, wie etwa $\mathbf{fun}(x{:}\,\mathbf{int})x + 1$ bekommt er eine Bedeutung und wird dann bei der Funktionsanwendung an das aktuale Argument gebunden.

Die binären Operationen haben die übliche Bedeutung, wobei $x - y = 0$, falls $x \leq y$ („monus“).

Bei der Fallunterscheidung hat 0 die Bedeutung von **true** und jeder andere Wert die Bedeutung von **false**.

Eine ähnliche Rolle wie die Abstraktion spielt die Konstante $\mathbf{fix}_\tau$. Mit ihr kann man *rekursive* Funktionen definieren, ohne sich einen neuen Namen zu spendieren. Würde man etwa intuitiv die Fakultät durch

$$\begin{array}{l}\mathbf{fact} \equiv \mathbf{fun}(x{:}\,\mathbf{int})\\ \qquad \mathbf{if}\; x \qquad (^*\; x = 0\; ^*)\\ \qquad\quad \mathbf{then}\; 1\\ \qquad\quad \mathbf{else}\; x \times \mathbf{fact}(x-1)\end{array}$$

definieren, so schreibt man offiziell

$$\begin{array}{l}\mathbf{fact} \equiv \mathbf{fix}_{\mathbf{int}\to\mathbf{int}}\, \mathbf{fun}(f{:}\,\mathbf{int} \to \mathbf{int})\\ \qquad \mathbf{fun}(x{:}\,\mathbf{int})\\ \qquad \mathbf{if}\; x \qquad (^*\; x = 0\; ^*)\\ \qquad\quad \mathbf{then}\; 1\\ \qquad\quad \mathbf{else}\; x \times f(x-1)\end{array}$$

Man fasst also eine rekursiv definierte Funktion auf als Fixpunkt des Funktionals, welches die Abhängigkeit der Funktion von sich selber beschreibt.

Nach dieser informellen Einführung jetzt die formalen Typisierungsregeln. Wir müssen die Typen von Variablen, die ja durch die Annotate in Funktionsabstraktionen gegeben sind, irgendwie mitführen und verwenden dazu einen sogenannten Kontext.

Solch ein Kontext Γ ist eine partielle Funktion von Variablen nach Typen. Hingeschrieben wird ein Kontext in der Form $x_1{:}\,\tau_1, x_2{:}\,\tau_2, \ldots, x_n{:}\,\tau_n$. Wenn Γ ein Kontext ist, so bezeichnet $\mathrm{dom}(\Gamma)$ die Menge der Variablen, denen von Γ ein Typ zugewiesen wird. Wenn x eine Variable und τ ein Typ ist, dann bezeichnet $\Gamma, x{:}\,\tau$ den Kontext, der zusätzlich zu den Zuweisungen von Γ noch x den Typ τ gibt; wurde x schon in Γ ein Typ zugewiesen, dann wird dies in $\Gamma, x{:}\,\tau$ „überschrieben". Den leeren Kontext, der keiner Variablen einen Typ zuweist, lässt man weg.

Das Typisierungsurteil $\Gamma \vdash e : \tau$, gelesen „e ist vom Typ τ im Kontext Γ", ist jetzt durch folgende Regeln definiert.

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\frac{n \in \mathbb{N}}{\Gamma \vdash n : \mathbf{int}}$$

$$\frac{\Gamma, x{:}\,\tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathbf{fun}(x{:}\,\tau_1)e : \tau_1 \to \tau_2}$$

$$\frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)}$$

$$\frac{}{\Gamma \vdash \mathbf{fix}_\tau : (\tau \to \tau) \to \tau}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{int} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathbf{if}\, e_1\, \mathbf{then}\, e_2\, \mathbf{else}\, e_3 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{int} \quad \Gamma \vdash e_2 : \mathbf{int} \quad \mathbf{op} \in \{+, -, \times\}}{\Gamma \vdash e_1\, \mathbf{op}\, e_2 : \mathbf{int}}$$

Sehen wir, wie $\mathbf{fun}(f{:}\,\mathbf{int} \to \mathbf{int})\mathbf{fun}(x{:}\,\mathbf{int})(f\,x) + 1$ den Typ $(\mathbf{int} \to \mathbf{int}) \to \mathbf{int} \to \mathbf{int}$ bekommt: Im Kontext $f{:}\,\mathbf{int} \to \mathbf{int}, x{:}\,\mathbf{int}$ ist $f : \mathbf{int} \to \mathbf{int}$ und

$x : \mathbf{int}$, also $f\,x : \mathbf{int}$ und außerdem $1 : \mathbf{int}$, somit $(f\ x) + 1 : \mathbf{int}$. Damit $\mathbf{fun}(x{:}\,\mathbf{int})(f\ x) + 1 : \mathbf{int} \to \mathbf{int}$ im Kontext $f{:}\,\mathbf{int} \to \mathbf{int}$ und schließlich $\mathbf{fun}(f{:}\,\mathbf{int} \to \mathbf{int})\mathbf{fun}(x{:}\,\mathbf{int})(f\ x) + 1 : \mathbf{int} \to \mathbf{int}$ im leeren Kontext.

Bei einem so einfachen Typsystem wie hier könnte man die Typisierung wahrscheinlich auch ganz gut mit Worten beschreiben. Der Vorteil der regelbasierten Definition der Typisierung liegt in der Erweiterbarkeit auf kompliziertere Typsysteme (etwa auf das von ML). Es ist daher gut, die Methode mal gesehen zu haben.

4.1.1 Freie Variablen und Substitution

Eine Variable x kommt frei in einem Term e vor, wenn sie nicht durch eine Abstraktion $\mathbf{fun}(x{:}\,\tau)$ gebunden wird. Z.B. kommt die Variable x frei vor in den Termen x, $\mathbf{fun}(y{:}\,\mathbf{int})x$ und $(\mathbf{fun}(x{:}\,\mathbf{int})x)x$, aber sie kommt nicht frei vor in den Termen 0, $\mathbf{fun}(x{:}\,\mathbf{int})x$, $\mathbf{fun}(y{:}\,\mathbf{int})\mathbf{fun}(x{:}\,\mathbf{int})x$. Formal definiert man die Menge $\mathsf{FV}(e)$ der freien Variablen des Terms e durch Induktion über den Aufbau:

$$
\begin{aligned}
&\mathsf{FV}(n) = \emptyset\\
&\mathsf{FV}(e_1\,e_2) = \mathsf{FV}(e_1) \cup \mathsf{FV}(e_2)\\
&\mathsf{FV}(\mathbf{fun}(x{:}\,\tau)e) = \mathsf{FV}(e) \setminus \{x\}\\
&\mathsf{FV}(x) = \{x\}\\
&\mathsf{FV}(\mathbf{fix}_\tau\,) = \emptyset\\
&\mathsf{FV}(e_1\ \mathbf{op}\ e_2) = \mathsf{FV}(e_1) \cup \mathsf{FV}(e_2)\\
&\mathsf{FV}(\mathbf{if}\,e_1\,\mathbf{then}\,e_2\,\mathbf{else}\,e_3) = \mathsf{FV}(e_1) \cup \mathsf{FV}(e_2) \cup \mathsf{FV}(e_3)
\end{aligned}
$$

Ein Term e ist *geschlossen*, wenn $\mathsf{FV}(e) = \emptyset$. Die freien Variablen kann man für jeden Pseudoterm definieren. Es gilt folgender Zusammenhang zwischen Typisierungen und freien Variablen, den man sofort durch Induktion über den Termaufbau einsieht.

Lemma 4.1 *Gilt $\Gamma \vdash e : \tau$, so ist $\mathsf{FV}(e)$ enthalten in $\mathrm{dom}(\Gamma)$, es wird also jede freie Variable von e in Γ erwähnt. Außerdem gilt dann $\Gamma \restriction \mathsf{FV}(e) \vdash e : \tau$*

Ist also e ein geschlossener, wohltypisierter Term, so gilt $\vdash e : \tau$ für einen Typ τ. Man sagt dann, e ist ein *Programm* vom Typ τ.

Ist x Variable, e ein beliebiger Term und e_1 geschlossen, so ist die Substitution von e_1 für x in e definiert als der Term $[e_1/x]e$, den man durch Ersetzung aller freien Vorkommen von x in e durch e_1 erhält. Z.B. ist $[103/x]x + 6 = 103 + 6$ und $[\mathbf{fun}(x{:}\,\mathbf{int})x/x](\mathbf{fun}(x{:}\,\mathbf{int}{\to}\mathbf{int})x)x = (\mathbf{fun}(x{:}\,\mathbf{int}{\to}\mathbf{int})x)(\mathbf{fun}(x{:}\,\mathbf{int})x)$. Formal definieren wir die Substitution durch Induktion über den Termaufbau.

Die Klauseln sind dann wie folgt:

$$
\begin{array}{lcl}
[e_0/x]n & = & n \\[1em]
[e_0/x]e_1\ e_2 & = & ([e_0/x]e_1)([e_0/x]e_2) \\[1em]
[e_0/x]\mathbf{fun}(y\colon\tau)e_1 & = & \begin{cases} \mathbf{fun}(y\colon\tau)e_1, \text{ falls } x = y \\ \mathbf{fun}(y\colon\tau)[e_0/x]e_1, \text{ falls } x \neq y \end{cases} \\[1em]
[e_0/x]y & = & \begin{cases} e_0, \text{ wenn } x = y \\ y, \text{ wenn } x \neq y \end{cases} \\[1em]
[e_0/x]\mathbf{fix}_\tau & = & \mathbf{fix}_\tau \\[1em]
[e_0/x](\mathbf{if}\, e_1\, \mathbf{then}\, e_2\, \mathbf{else}\, e_3) & = & \mathbf{if}\, [e_0/x]e_1\, \mathbf{then}\, [e_0/x]e_2\, \mathbf{else}\, [e_0/x]e_3 \\
[e_0/x](e_1\, \mathsf{op}\, e_2) & = & [e_0/x]e_1\, \mathsf{op}\, [e_0/x]e_2
\end{array}
$$

Die Substitution wird also rekursiv in einen Term hineingeschoben. Die substituierte Variable wird durch das Substitut ersetzt, andere Variablen bleiben unverändert. Tritt die zu substituierende Variable gebunden auf, so wird sie nicht substituiert.

Man beachte, dass wir nur geschlossene Terme substituieren; es besteht somit nicht die Gefahr unerwünschter Variablenbindungen, der man durch Umbenennung begegnen müsste.

Die Substitution hängt folgendermassen mit der Typisierung zusammen:

Lemma 4.2 *Wenn* $\Gamma, x\colon\tau_1 \vdash e : \tau$ *und* $\Gamma \vdash e_1 : \tau_1$, *so gilt* $\Gamma \vdash [e_1/x]e : \tau$.

4.2 Operationale Semantik

Die operationale Semantik von PCF legt fest, wie ein geschlossener Term ausgewertet wird. Intuitiv geht man dabei folgendermaßen vor. Funktionsanwendungen $(\mathbf{fun}(x\colon\tau)e_1)e_2$ werden durch β-Reduktion ausgewertet, man setzt also das aktuale Argument e_2 für den formalen Parameter x ein. Binäre Operationen e_1 **op** e_2 werden ausgewertet, indem man e_1 und e_2 auswertet, bzw. wenn diese bereits Konstanten sind, die entsprechende Operation durchführt. Bei der Fallunterscheidung wertet man zunächst nur e_1 aus; wenn das eine Konstante liefert, dann sucht man den ensprechenden Zweig raus; eine rekursive Definition $\mathbf{fix}_\tau\ e$ schließlich ersetzt man durch $e\ (\mathbf{fix}_\tau\ e)$. Das kann natürlich nur dann terminieren, wenn das e irgendwann sein Argument nicht mehr anguckt. Dieses „nicht mehr Angucken“ passiert meistens durch Verwendung der Fallunterscheidung, die ja jeweils einen Zweig unausgewertet verwirft.

Formal definieren wir eine Auswerterelation $\Rightarrow$ zwischen geschlossenen Termen durch Induktion über den Aufbau. Die Regeln sind wie folgt.

$$\text{(E-}\textsc{App}\text{-L)}\quad \frac{e_1 \Rightarrow e_1'}{e_1 e_2 \Rightarrow e_1' e_2}$$

$$\text{(E-}\textsc{App}\text{-B)}\quad \frac{}{(\mathbf{fun}(x{:}\,\tau)e_1)\ e_2 \Rightarrow [e_2/x]e_1}$$

$$\text{(E-}\textsc{Bin}\text{-L)}\quad \frac{e_1 \Rightarrow e_1'}{e_1\,\mathbf{op} e_2 \Rightarrow e_1'\,\mathbf{op}\,e_2}$$

$$\text{(E-}\textsc{Bin}\text{-R)}\quad \frac{e_2 \Rightarrow e_2'}{n\,\mathbf{op} e_2 \Rightarrow n\,\mathbf{op}\,e_2'}$$

$$\text{(E-}\textsc{Bin}\text{-B)}\quad \frac{n\,\mathbf{op}\,m}{k},\ \text{wobei } k = n\,\mathbf{op}\,m$$

$$\text{(E-}\textsc{If}\text{-L)}\quad \frac{e_1 \Rightarrow e_1'}{\mathbf{if}\,e_1\,\mathbf{then}\,e_2\,\mathbf{else}\,e_3 \Rightarrow \mathbf{if}\,e_1'\,\mathbf{then}\,e_2\,\mathbf{else}\,e_3}$$

$$\text{(E-}\textsc{If}\text{-B-0)}\quad \frac{}{\mathbf{if}\,0\,\mathbf{then}\,e_2\,\mathbf{else}\,e_3 \Rightarrow e_2}$$

$$\text{(E-}\textsc{If}\text{-B-1)}\quad \frac{}{\mathbf{if}\,n\,\mathbf{then}\,e_2\,\mathbf{else}\,e_3 \Rightarrow e_3},\ \text{falls } n \neq 0$$

$$\text{(E-}\textsc{Fix}\text{)}\quad \frac{}{\mathbf{fix}_\tau\ \ e \Rightarrow e\ (\mathbf{fix}_\tau\ \ e)}$$

Es ist klar, dass auf einen beliebigen Pseudoterm immer höchstens eine Regel anwendbar ist, also folgt aus $e \Rightarrow e_1$ und $e \Rightarrow e_2$, dass $e_1 = e_2$. Ein Term e wird nun dadurch ausgewertet, dass man mit Hilfe der Regeln eine Folge $e \Rightarrow e_1 \Rightarrow e_2 \Rightarrow \ldots$ konstruiert. Ist diese Folge endlich; wird also ein Term e' erreicht, der nicht weiter reduziert werden kann, also z.B. eine Konstante oder eine Funktionsabstraktion, so sagen wir e *hat Wert* e'. Ist die Folge unendlich, so sagen wir e *divergiert* oder e terminiert nicht.

4.2.1 Beispiele

Eine einfache β-Reduktion:

$$\begin{aligned}
&(\mathbf{fun}(f\colon \mathbf{int} \to \mathbf{int})\mathbf{fun}(x\colon \mathbf{int})f(x+1))(\mathbf{fun}(x\colon \mathbf{int})x+x)6 \Rightarrow\\
&(\mathbf{fun}(x\colon \mathbf{int})(\mathbf{fun}(x\colon \mathbf{int})x+x)(x+1))6 \Rightarrow\\
&(\mathbf{fun}(x\colon \mathbf{int})x+x)(6+1) \Rightarrow (6+1)+(6+1) \Rightarrow\\
&7+(6+1) \Rightarrow 7+7 \Rightarrow 14
\end{aligned}$$

Für jeden Typ τ hat man ein nichtterminierendes Programm $\Omega_\tau := \mathbf{fix}_\tau\,(\mathbf{fun}(x\colon \tau)x)$. Es gilt

$$\Omega_\tau \Rightarrow (\mathbf{fun}(x\colon \tau)x)\Omega_\tau \Rightarrow \Omega_\tau \Rightarrow \ \ldots$$

Natürlich gibt es auch terminierende Programme mit **fix** . Zum Beispiel haben wir

$$\mathbf{fix}_{\mathbf{int}}\,(\mathbf{fun}(x\colon \mathbf{int})5) \Rightarrow (\mathbf{fun}(x\colon \mathbf{int})5)(\mathbf{fix}_{\mathbf{int}}\,(\mathbf{fun}(x\colon \mathbf{int})5)) \Rightarrow 5$$

Oder auch etwas interessanter für $e \equiv \mathbf{fix}_{\mathbf{int}\to\mathbf{int}}\,(\mathbf{fun}(f\colon \mathbf{int} \to \mathbf{int})\mathbf{fun}(x\colon \mathbf{int})\mathbf{if}\ x\ \mathbf{then}\ 0\ \mathbf{else}\ x+f(x-1)$:

$$\begin{aligned}
&e\ 1 \Rightarrow\\
&(\mathbf{fun}(f\colon \mathbf{int} \to \mathbf{int})\mathbf{fun}(x\colon \mathbf{int})\mathbf{if}\ x\ \mathbf{then}\ 0\ \mathbf{else}\ x+f(x-1))\ e\ 1 \Rightarrow\\
&(\mathbf{fun}(x\colon \mathbf{int})\mathbf{if}\ x\ \mathbf{then}\ 0\ \mathbf{else}\ x+e(x-1))\ 1 \Rightarrow\\
&\mathbf{if}\ 1\ \mathbf{then}\ 0\ \mathbf{else}\ 1+e(1-1) \Rightarrow\\
&1+e(1-1) \Rightarrow \ldots\\
&1+\mathbf{if}\ 1-1\ \mathbf{then}\ 0\ \mathbf{else}\ e(1-1-1) \Rightarrow \ldots\\
&1+\mathbf{if}\ 0\ \mathbf{then}\ 0\ \mathbf{else}\ e(1-1-1) \Rightarrow 1+0 \Rightarrow 1
\end{aligned}$$

Auch Programme, die Ω_τ enthalten können terminieren:

$$(\mathbf{fun}(x\colon \mathbf{int})5)\Omega_{\mathbf{int}} \Rightarrow 5$$

4.2.2 Type soundness

Eine wichtige Eigenschaft der Reduktion ist, dass sie die Typen erhält:

Satz 4.3 *Gilt* $\vdash e : \tau$ *und* $e \Rightarrow e'$, *so auch* $\vdash e' : \tau$.

Beweis. Durch Induktion über die Herleitung von $\vdash e : \tau$ unter Verwendung von Lemma 4.2.

Diese Eigenschaft nennt man in der engl. Literatur *subject reduction* bezeichnet. Als wichtige Folgerung ergibt sich die sog. *type soundness*:

Satz 4.4 *Ist e wohlgeformter geschlossener Term, also $\vdash e : \tau$, dann hat entweder e eine Konstante oder eine Abstraktion als Wert, oder e divergiert.*

Type soundness bedeutet also, dass die Reduktion eines wohltypisierten Terms nicht zu einem "Laufzeit-Typfehler" wie $10\,(3)$ oder $(\mathbf{fun}(x{:}\,\mathbf{int})x) + 13$ führen kann.Der Turingpreisträger Robin Milner hat den griffigen Slogan *well typed programs cannot go wrong* für diese Eigenschaft geprägt.

Bei PCF ist *type soundness* eine relativ oberflächliche Eigenschaft. Interessanter wird es bei objektorientierten Programmiersprachen. Hier kann ein korrektes Typsystem Laufzeitfehler ausschließen, die auf nicht implementierte Methoden zurückzuführen sind.

4.2.3 Big step und small step

Die obige operationale Semantik für PCF ist eine *small step* Semantik, da sie die Auswertung eines Terms in eine Folge atomarer Schritte zerlegt. Die operationale Semantik von IMP dagegen war eine *big step* Semantik, da sie das endgültige Resultat einer Rechnung gleich in einem Schritt liefert. Auch für PCF kann man eine *big step* Semantik definieren. Das wäre dann ein induktiv definiertes Urteil der Form $e \Longrightarrow e'$, wobei e ein Ausdruck ist und e' ein „Wert", also eine Konstante oder ein Abstraktion. Durch Induktion könnte man dann nachweisen, dass $e \Rightarrow^* e'$ für einen Wert e' genau dann, wenn $e \Longrightarrow e'$. Ob jetzt *big step* oder *small step* die bessere Beschreibung darstellt, darüber scheiden sich die Geister. Manchmal erlaubt die *big step* Semantik etwas kürzere Beweise, auf der anderen Seite ist die *small step* Semantik robuster gegenüber Spracherweiterungen, etwa um Parallelität und Nichtdeterminismus.

4.2.4 Lazy vs. eager

Eine fundamentalere Eigenschaft der operationalen Semantik ist, dass sie festlegt, dass in einer Applikation das Argument direkt für den formalen Parameter eingesetzt wird und nicht erst ausgewertet wird. D.h. es wird erst dann ausgewertet, wenn es wirklich sein muss, also wenn eine Rechenoperation darauf angewendet werden soll, es selbst eine Funktion ist, die auf ein Argument angewendet wird, oder es als erstes Argument einer Fallunterscheidung erscheint.

Die Semantik wird daher als *lazy* oder *call by name* bezeichnet. So eine *lazy* Semantik haben auch die funktionalen Programmiersprachen Haskell und Miranda

Alternativ kann man auch eine *eager* oder *call by value* Semantik betrachten, in der eine Funktionsanwendung erst dann reduziert wird, wenn das Argument voll ausgewertet ist. Solch eine Semantik haben etwa ML und LISP. Die *eager*

Semantik hat den Vorteil, dass sie im allgemeinen effizienter zu implementieren ist und dass die *lazy* Semantik in ihr codiert werden kann. Nachteile der *eager* Semantik sind vor allem ihre etwas kompliziertere Formulierung (zum Beispiel führt unter *call by value* die Regel E-FIX sofort und immer in eine Endlosschleife und muss daher geeignet modifiziert werden).

4.3 Beobachtungsgleichheit

Wie kann man PCF Programme spezifizieren und verifizieren? Im Gegensatz zu IMP gibt es in PCF keinen Zustand, also braucht man auch keine Zusicherungen. Dafür haben jetzt Programme einen Wert (etwa eine Konstante oder eine Funktion) und über diesen Wert möchte man in einer Spezifikation reden. Die elementarste Art, das zu tun, besteht aus Gleichheitsaussagen. Aus Gleichheitsaussagen kann man dann durch logische Konnektive und Quantifizierung kompliziertere Spezifikationen aufbauen. Die Fakultätsfunktion möchte man etwa durch

$$\forall n {\in} \mathbf{N}.(\mathsf{fact}\ n) = n! \qquad (\dagger)$$

spezifizieren.

Was soll nun diese Gleichheit bedeuten? Syntaktische Gleichheit von Termen ist sicher nicht adäquat, denn dann wäre (†) nicht erfüllt, denn ($\mathsf{fact}\ n$) ist eine Applikation und $n!$ ist eine Konstante. Besser ist folgende auf der Reduktion basierende Gleichheit:

Definition 4.5 *Zwei Programme e, e' vom Typ* **int** *heißen* operational äquivalent*, in Zeichen $e \simeq_{\mathbf{int}} e'$, wenn für alle $n \in \mathbf{N}$ gilt: $e \Rightarrow^* n$ genau dann, wenn $e' \Rightarrow^* n$.*

Mit anderen Worten bedeutet $e \simeq_{\mathbf{int}} e'$, dass entweder e, e' beide divergieren, oder beide denselben Wert haben.

Wie sieht es nun mit höheren Typen aus? Wann sollen zwei Programme vom Typ **int** $\rightarrow$ **int** als gleich gelten? Wenn man obige Definition naiv verallgemeinert, dann wären zum Beispiel $\mathbf{fun}(x{:}\,\mathbf{int})x$ und $\mathbf{fun}(x{:}\,\mathbf{int})x + 1$ verschieden, da ja beide nicht identisch sind, aber auch nicht weiter reduzieren. Es gibt zwei Möglichkeiten, Gleichheit von Funktionen sinnvoll zu definieren und wir werden später sehen, dass für PCF beide sogar übereinstimmen. Der Beweis dieser Tatsache ist aber nicht ganz einfach.

Der eine Gleichheitsbegriff, die *Beobachtungsgleichheit* geht davon aus, dass alle wirklich ausgeführten Programme vom Typ **int** sind; die anderen treten nur als Unterprogramme von solchen auf. Daher kann man sagen, dass zwei Programme e, e' von beliebigem Typ τ als gleich gelten können, wenn sie in ihrer Rolle als Unterprogramme eines grossen Programmes vom Typ **int** beliebig austauschbar sind.

Definition 4.6 *Seien e, e' Programme eines (gemeinsamen) Typs τ. Wir sagen e, e' sind* beobachtungsgleich*, i.Z. $e =^{\mathrm{obs}}_{\tau} e'$, wenn für alle Programme $f : \tau \to \mathbf{int}$ gilt $f\ e \simeq_{\mathbf{int}} f\ e'$.*

Man beachte, dass hier die Idee, dass e, e' als Unterprogramme vorkommen durch Funktionsapplikation modelliert wird. Man könnte auch festlegen, dass $e =^{\mathrm{obs}}_{\tau} e'$, wenn für alle Terme f mit $x{:}\,\mathbf{int} \vdash f : \tau$ gilt $[e/x]f \simeq_{\mathbf{int}} [e'/x]f$. Es ist aber leicht zu sehen, dass diese Definition äquivalent ist.

Die Beobachtungsgleichheit hat neben der, wie ich meine, guten Motivation den Vorteil, mit allen Termformern (außer der Abstraktion) verträglich zu sein:

Lemma 4.7 *Wenn $e_1 =^{\mathrm{obs}}_{\tau \to \tau'} e_1'$ und $e_2 =^{\mathrm{obs}}_{\tau} e_2'$, dann auch $e_1 e_2 =^{\mathrm{obs}}_{\tau'} e_1' e_2'$.*

Beweis. Übung.

Verträglichkeit mit der Abstraktion ist nicht sinnvoll, da ja die Beobachtungsgleichheit nur für Programme (also Terme ohne freie Variablen) erklärt ist. Man kann die Beobachtungsgleicheit auch für Terme mit freien Variablen definieren, indem man von allen freien Variablen abstrahiert. Die so erhaltene Gleichheit ist dann per Definition mit der Abstraktion verträglich.

Der große Nachteil der Beobachtungsgleichheit ist, dass sie ohne weitere Hilfsmittel ziemlich schwer nachzuweisen ist. Wie soll man etwa einsehen, dass $\mathbf{fun}(x{:}\,\mathbf{int})x$ und $\mathbf{fun}(x{:}\,\mathbf{int})x+0$ beobachtungsgleich sind? Gewiss bezeichnen die beiden Programme dieselbe Funktion, aber es könnte ja doch einen sehr komplizierten irgendwie rekursiv definierten Kontext geben, in dem sich die beiden unterscheiden.

Dass dem nicht so ist, werden wir aus einem allgemeineren Resultat ableiten; um das aber überhaupt formulieren zu können, brauchen wir einen anderen Gleichheitsbegriff, der eigentlich auch ganz gut motiviert ist: Programme funktionalen Typs sollen als gleich gelten, wenn sie für alle Argumente das gleiche Ergebnis liefern. Dies sieht zunächst zirkulär aus, aber letzendlich ist das Ergebnis einer jeden Funktion ja vom Typ **int**, sodass wir nur die schon definierte operationale Äquivalenz auf Typ **int** verwenden müssen.

Definition 4.8 *Zwei Programme e, e' vom Typ $\tau = \tau_1 \to \tau_2 \to \ldots \to \tau_n \to \mathbf{int}$ sind* operational äquivalent*, i.Z. $e \simeq_{\tau} e'$, wenn für alle Programme $e_1{:}\,\tau_1, e_2{:}\,\tau_2, \ldots, e_n{:}\,\tau_n$ gilt*

$$e\ e_1\ e_2\ \ldots\ e_n \simeq_{\mathbf{int}} e'\ e_1\ e_2\ \ldots\ e_n$$

Man beachte, dass sich jeder Typ τ in der Form $\tau = \tau_1 \to \tau_2 \to \ldots \to \tau_n \to \tau$ schreiben lässt, sodass diese Definition vollständig ist.

Als Beispiel zeigen wir, dass

$$e := \mathbf{fun}(x{:}\,\mathbf{int})x \simeq_{\mathbf{int}\to\mathbf{int}} \mathbf{fun}(x{:}\,\mathbf{int})x + 0 =: e'$$

Sei $e_1 : \mathbf{int}$ beliebig vorgegeben. Wenn e_1 divergiert, dann divergieren sowohl $e\ e_1$, als auch $e'\ e_1$. Hat dagegen e_1 den Wert n, so haben sowohl $e\ e_1$, als auch $e\ e_2$ diesen Wert.

Man sieht, dass sich die operationale Äquivalenz relativ leicht nachweisen lässt. Ihr Nachteil neben der etwas schlechteren Motivation ist, dass a priori kein Grund besteht, warum sie mit den anderen Termformern verträglich sein sollte. D.h. wenn $e \simeq_\tau e'$ und $f : \tau \to \tau'$ beliebig ist, so ist nicht klar, wieso $f\ e \simeq_{\tau'} f\ e'$ gelten sollte. Man müsste dazu wissen, dass f sein Argument nur auf geschlossene Terme anwenden, denn für die weiss man ja aufgrund der Definition von $\simeq$, dass e und e' gleich sind. Das muss aber nicht so sein. Zum Beispiel wenn f gleich **fix** ist.

Es fällt nun schwer, hier weiter herumzuphilosophieren, da die operationale Äquivalenz eben doch mit allen Termformern verträglich ist und zwar ganz einfach deshalb, weil sie mit der Beobachtungsgleichheit koinzidiert! Diese als „Milners's Kontextlemma“ bezeichnete Eigenschaft kann man zwar grundsätzlich syntaktisch zeigen. Transparenter ist aber ein Beweis, der auf einer denotationalen Semantik für PCF beruht, der wir uns im folgenden zuwenden wollen.

5 Denotationale Semantik

In diesem Abschnitt wollen wir eine Semantik von PCF kennenlernen, in der Ausdrücke wirkliche Funktionen bedeuten und die Applikation wirkliche Funktionsanwendung. Diese sog. *denotationale Semantik* werden wir dann mit der operationalen Semantik so in Beziehung setzen, dass wir sie verwenden können, um bestimmte Regeln für die Beobachtungsgleichheit herzuleiten. Insbesondere werden wir Milner's Kontextlemma zeigen und eine Regel für die Verifikation rekursiver Programme herleiten.

Ein naiver Versuch besteht darin, Typen als Mengen zu interpretieren, also etwa $[\![\mathbf{int}]\!] = \mathbf{N}$ und Funktionstypen als Funktionenmengen, also etwa $[\![\tau \to \tau']\!] =$ „Menge der Funktionen von $[\![\tau]\!]$ nach $[\![\tau']\!]$“.

Damit kann man die Abstraktion und die Applikation interpretieren; nicht aber die rekursive Definition. Die Nachfolgerfunktion $\mathsf{succ} = \lambda x{\in}\mathbf{N}.x + 1$ hat nun mal keinen Fixpunkt, aber das Programm $\mathbf{fix_{int}}\,(\mathbf{fun}(x{:}\,\mathbf{int})x + 1)$ gibt es, und es soll irgendwie interpretiert werden. Man könnte sagen, dass dieses Programm nicht terminiert und es deshalb auch keine Semantik bekommen soll. Aber was macht man dann mit Programmen wie $e \equiv \mathbf{fun}(x{:}\,\mathbf{int})\mathbf{if}\ x\ \mathbf{then}\ 0\ \mathbf{else}\ \Omega_{\mathbf{int}}$? Das terminiert auf jeden Fall als Funktion und auch angewendet auf 0. Nur die anderen Anwendungen $e\ n$ für $n \neq 0$ divergieren. Der Term e bezeichnet also eine *partielle* Funktion von $\mathbf{N}$ nach $\mathbf{N}$. Partielle Funktionen sind nun aber nicht fein genug, denn die beiden Programme $e_1 = \lambda x{:}\,\mathbf{int}.0$ und $e_2 = \lambda x{:}\,\mathbf{int}.\mathbf{if}\ x\ \mathbf{then}\ 0\ \mathbf{else}\ 0$ bezeichnen dieselbe (partielle) Funktion, sind aber nicht beobachtungsgleich, weil $e_1\Omega_{\mathbf{int}} \Rightarrow^* 0$ und $e_2\Omega_{\mathbf{int}}$ divergiert.

Die Semantik sollte also das Verhalten bezüglich undefinierter Elemente mitberücksichtigen. Das legt nahe, die Undefiniertheit semantisch durch ein spezielles Element $\perp$ (genannt *bottom*) zu modellieren. Wir interpretieren also den Basistyp **int** als die Menge $\mathbf{N}_\perp := \mathbf{N} \cup \{\perp\}$. Wenn wir nun Funktionen vom Typ $\mathbf{int} \to \mathbf{int}$ als beliebige Funktionen von $\mathbf{N}_\perp$ nach $\mathbf{N}_\perp$ interpretieren, dann können wir wieder die Rekursion nicht gut modellieren, denn die Nachfolgerfunktion succ erweitert um $\mathsf{succ}(\perp) = 0$ hat auch keinen Fixpunkt.

Hier kommt nun folgende Intuition über $\perp$ ins Spiel. Wenn ein Programm angewendet auf ein nichtterminierendes Argument, also etwa Ω terminiert, dann nur deshalb, weil das Argument gar nicht angeschaut wurde, wie etwa in $\mathbf{fun}(x\colon \mathbf{int})5$. Dann muss aber die Funktion für alle anderen Argumente dasselbe Resultat liefern.

Eine PCF-definierbare Funktion bezeichnet also eine Funktion von $\mathbf{N}_\perp$ nach $\mathbf{N}_\perp$, die *monoton* ist bezüglich der durch

$$x \sqsubseteq y \Leftrightarrow x = \perp \vee x = y$$

definierten Ordnung auf $\mathbf{N}_\perp$.

Jede monotone Funktion $f : \mathbf{N}_\perp \to \mathbf{N}_\perp$ hat in der Tat einen Fixpunkt, nämlich $f(\perp)$.

Man bezeichnet die Ordnung auf $\mathbf{N}_\perp$ als *Informationsordnung*. Eine konkrete Zahl n beinhaltet mehr Information als $\perp$ und eine monotone Funktion reagiert auf Erhöhung des Informationsgehaltes der Eingabe nicht mit einer Verkleinerung des Informationsgehaltes der Ausgabe.

Den Typ $\mathbf{int} \to \mathbf{int}$ interpretieren wir also als die Menge $\mathbf{N}_\perp \Rightarrow \mathbf{N}_\perp$ der monotonen Funktionen von $\mathbf{N}_\perp$ nach $\mathbf{N}_\perp$. Wie sieht es mit Funktionalen $(\mathbf{int} \to \mathbf{int}) \to (\mathbf{int} \to \mathbf{int})$ aus, die ja die typischen Argumente für **fix** bilden? Auf jeden Fall sollten die monoton sein bezüglich der durch

$$f \sqsubseteq g \Leftrightarrow \forall x {\in} \mathbf{N}_\perp . f(x) \sqsubseteq f(y)$$

definierten punktweisen Ordnung auf $\mathbf{N}_\perp \to \mathbf{N}_\perp$.

Schreiben wir $\mathrm{dom}(f)$ für $\{x \mid f(x) \neq \perp\}$, so gilt, dass $f \sqsubseteq g$ genau dann, wenn $\mathrm{dom}(f) \subseteq \mathrm{dom}(g)$ und $f(x) = g(x)$ für alle $x \in \mathrm{dom}(f)$.

Man kann zeigen, dass jede monotone Funktion von $\mathbf{N}_\perp \Rightarrow \mathbf{N}_\perp$ nach $\mathbf{N}_\perp \Rightarrow \mathbf{N}_\perp$ einen kleinsten Fixpunkt besitzt und man könnte das auch auf höhere Typen fortsetzen. Die denotationelle Semantik wird aber genauer, wenn man noch einen weiteren Gesichtspunkt der operationalen Semantik mit einbezieht. Jedes PCF definierbare Funktional $f : (\mathbf{int} \to \mathbf{int}) \to \mathbf{int} \to \mathbf{int}$ hat nämlich die Eigenschaft, dass in $f\ u\ n$ das Argument $u : \mathbf{int} \to \mathbf{int}$ nur an endlich vielen (von n und dem Werteverlauf von u abhängigen) Argumenten aufgerufen wird.

Das kann man im folgenden Sinne als „Stetigkeit“ deuten: Wenn $f_1 \sqsubseteq f_2 \sqsubseteq \dots$ eine aufsteigende Kette von Funktionen in $\mathbf{N}_\perp \Rightarrow \mathbf{N}_\perp$ ist, also $\mathrm{dom}(f_{i+1}) \supseteq$

$\mathrm{dom}(f_i)$ und $f_{i+1}(x) = f_i(x)$ für alle $x \in \mathrm{dom}(f_i)$, dann wird durch

$$\bigsqcup_{i \geq 0} f_i := \lambda x {\in} \mathbf{N}_\perp . \begin{cases} z, \text{ falls } \exists i . f_i(x) = z \neq \perp \\ \perp, \text{ sonst} \end{cases}$$

das Supremum der f_i in der partiellen Ordnung $\mathbf{N}_\perp \Rightarrow \mathbf{N}_\perp$. Eine monotone Funktion $F : (\mathbf{N}_\perp \Rightarrow \mathbf{N}_\perp) \Rightarrow (\mathbf{N}_\perp \Rightarrow \mathbf{N}_\perp)$, die von einem PCF-Term herrührt, hat nun die Eigenschaft, solche Suprema zu erhalten: Es gilt

$$F(\bigsqcup_{i \geq 0} f_i) = \bigsqcup_{i \geq 0} F(f_i)$$

denn zur Berechnung von $F(\bigsqcup_{i \geq 0} f_i)(n)$ wurden nur endlich viele Werte von $\bigsqcup_{i \geq 0} f_i$ verwendet; die aber stellt bereits ein f_i für genügend großes i zur Verfügung.

5.1 Domains

Dies mag die folgenden Begriffsbildungen motivieren, die von den Spezialfällen $\mathbf{N}_\perp$ und $\mathbf{N}_\perp \Rightarrow \mathbf{N}_\perp$ abstrahieren und einen allgemeinen Rahmen zur Interpretation von Typen und Termen funktionaler Sprachen bereitstellen:

Sei $(D, \sqsubseteq)$ eine partielle Ordnung und

$$d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \dots$$

eine aufsteigende Kette von Elementen in D. Ein Element $e \in D$ heißt Supremum oder kleinste obere Schranke der Kette, wenn $e \sqsupseteq d_i$ für alle i und außerdem aus $e' \sqsupseteq d_i$ für alle i folgt, dass $e' \sqsupseteq e$.

So ein Supremum muss es nicht unbedingt geben, z.B. gibt es in $D = \mathbf{N}$ mit der natürlichen Ordnung $\sqsubseteq = \leq$ die Kette $0 \leq 1 \leq 2 \leq \dots$, die kein Supremum besitzt. Wenn es aber ein Supremum gibt, dann ist es eindeutig bestimmt, was die Notation $\bigsqcup_{i \geq 0} d_i$ für das Supremum rechtfertigt.

Ein Element $e \in D$ ist das kleinste, wenn $e \sqsubseteq e'$ für alle $e' \in D$. Auch das kleinste Element ist wenn es existiert eindeutig bestimmt, was die Notation $\perp_D$ oder kurz $\perp$ rechtfertigt.

Definition 5.1 *Ein partiellen Ordnung $(D, \sqsubseteq)$ ist ein* Domain *(auf deutsch kann man auch „Bereich“ sagen), wenn D ein kleinstes Element hat und jede aufsteigende Kette ein Supremum besitzt.*

Beispiele von Domains sind $\mathbf{N}_\perp$ und $\mathbf{N}_\perp \Rightarrow \mathbf{N}_\perp$, aber auch $\mathbf{N} \cup \{\infty\}$ mit der natürlichen Ordnung erweitert um $\infty \geq n$. Beachte, dass in $\mathbf{N} \cup \{\infty\}$ Suprema beliebiger Teilmengen existieren, in $\mathbf{N}_\perp$ und $\mathbf{N}_\perp \Rightarrow \mathbf{N}_\perp$ aber nicht. Zum Beispiel hat die Menge $\{0, 1\}$ in $\mathbf{N}_\perp$ kein Supremum.

Definition 5.2 *Seien D, E Domains. Eine Funktion $f : D \to E$ ist stetig, wenn sie monoton ist und Suprema aufsteigender Ketten erhält, d.h. wenn für jede aufsteigende Kette $d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \dots$ in D gilt*

$$f(\bigsqcup_{i\geq 0} d_i) = \bigsqcup_{i\geq 0} f(d_i)$$

Beachte, dass wegen der Monotonie die Kette $f(d_0) \sqsubseteq f(d_1) \sqsubseteq \dots$ aufsteigend ist; ihr Supremum also existiert. Beachte auch, dass schon aufgrund der Monotonie gilt $\bigsqcup_{i\geq 0} f(d_i) \sqsubseteq f(\bigsqcup_{i\geq 0} d_i)$.

Jede konstante Funktion ist stetig. Für jedes $x \in \mathbf{N}_\perp$ ist das Auswertefunktional $\lambda f.f(x) : (\mathbf{N}_\perp \Rightarrow \mathbf{N}_\perp) \to \mathbf{N}_\perp$ stetig. Auch das „Fakultätsfunktional" $F_{\mathsf{fact}} : (\mathbf{N}_\perp \Rightarrow \mathbf{N}_\perp) \longrightarrow (\mathbf{N}_\perp \Rightarrow \mathbf{N}_\perp)$ definiert durch

$$F(f) = \lambda x{\in}\mathbf{N}_\perp. \begin{cases} \perp, \text{ falls } x = \perp \\ 1, \text{ falls } x = 0 \\ f(x-1), \text{ falls } x > 0 \end{cases}$$

ist stetig.

Bemerkung 5.3 Eine Teilmenge $U \subseteq D$ eines Domains D, die nach unten abgeschlossen ist und stetig in dem Sinne, dass mit einer Kette auch ihr Supremum in U liegt, nennt man *Scott-offen*. Mit den Scott-offenen Mengen wird jeder Domain zu einem topologischen Raum und der domaintheoretische Stetigkeitsbegriff fällt bzgl. dieser Topologie mit dem üblichen topologischen Stetigkeitsbegriff zusammen („Urbild einer offenen Menge ist offen").

Wenn D, E Domains sind, so bezeichnen wir mit $D \Rightarrow E$ die Menge der stetigen Funktionen von D nach E mit der punktweisen Ordnung, also $f \sqsubseteq g \Leftrightarrow \forall x{\in}D. f(x) \sqsubseteq g(x)$. Mit $D{\times}E$ bezeichnen wir die Menge $\{(d,e) \mid d{\in}D, e{\in}E\} \cup \{\perp\}$ zusammen mit der durch $\perp \sqsubseteq x$ und $(d,e) \sqsubseteq (d',e') \Leftrightarrow d \sqsubseteq d' \wedge e \sqsubseteq e'$ erklärten komponentenweisen Ordnung.

Lemma 5.4 *Wenn D, E Domains sind, so auch $D \Rightarrow E$ und $D \times E$.*

Ihre eigentliche Bedeutung erhalten die stetigen Funktionen aus der Tatsache, dass sie stets einen kleinsten Fixpunkt besitzen, den man dann zur Interpretation von **fix** verwenden kann.

Lemma 5.5 *Sei D Domain und $f : D \longrightarrow D$ stetig. Das mit $Y(f)$ bezeichnete mit $\mathsf{Y}(f)$ bezeichnete Supremum der Kette*

$$\perp \sqsubseteq f(\perp) \sqsubseteq f(f(\perp)) \sqsubseteq \dots f^i(\perp) \dots$$

ist der kleinste Fixpunkt der Funktion f.

Beweis. Wegen der Stetigkeit von f ist $f(\mathsf{Y}(f)) = \bigsqcup_{i\geq 0} f(f^i(\bot)) = \bigsqcup_{i\geq 0} f^{i+1}(\bot) = \bigsqcup_{i>0} f^i(\bot) = \bigsqcup_{i\geq 0} f^i(\bot) = \mathsf{Y}(f)$. Im letzten Schritt haben wir verwendet, dass sich das Supremum einer Kette nicht ändert, wenn man vorne noch ein $\bot$ dranhängt.

Ist $f(x) \sqsubseteq x$, so gilt $x \sqsubseteq f^i(\bot)$ für alle i, wie man sofort durch Induktion über i einsieht. Also ist dann $x \sqsupseteq \mathsf{Y}(f)$.

Das Element $\mathsf{Y}(f)$ ist also nicht nur der kleinste Fixpunkt von f, sondern sogar der kleinste „Präfixpunkt“, also das kleinste Element x mit $f(x) \sqsubseteq x$.

5.2 Interpretation von PCF

Wir können nun eine Interpretation von PCF Typen als Domains und von PCF Termen als Elemente dieser Domains definieren. Die Interpretation der Typen ist definiert durch

$$
\begin{aligned}
&[\![\mathbf{int}]\!] = \mathbf{N}_\bot \\
&[\![\tau \to \tau']\!] = [\![\tau]\!] \Rightarrow [\![\tau']\!]
\end{aligned}
$$

Für die Interpretation der Terme führen wir zunächst den Begriff der Variablenbelegung ein.

Definition 5.6 *Ist Γ ein Kontext, so ist eine Variablenbelegung für Γ eine Funktion η, die jeder in Γ erwähnten Variablen x ein Element $\eta(x)$ von $[\![\Gamma(x)]\!]$ zuweist.*

Ist nun e ein Term mit $\Gamma \vdash e : \tau$ und η eine Variablenbelegung für Γ, so können wir ein Element $[\![e]\!]\eta$ von $[\![\tau]\!]$ definieren—die Semantik von e. Die Variablenbelegungen lassen wir weg, sodass wir die Semantik eines *Programms* einfach als $[\![e]\!]$ schreiben. Die Definition der Semantik erfolgt durch Induktion über den Termaufbau. Die Klauseln sind wie folgt.

$$
\begin{array}{lcl}
[\![n]\!]\eta & = & n \\
[\![e_1\, e_2]\!]\eta & = & [\![e_1]\!]\eta([\![e_2]\!]\eta) \\
[\![\mathbf{fun}(x{:}\,\tau)e]\!]\eta & = & \lambda v{\in}[\![\tau]\!].[\![e]\!]\eta[x \mapsto v] \\
[\![x]\!]\eta & = & \eta(x) \\
[\![\mathbf{if}\; e_1\; \mathbf{then}\; e_2\; \mathbf{else}\; e_3]\!]\eta & = & \begin{cases} \bot, \text{ falls } [\![e_1]\!]\eta = \bot \\ [\![e_2]\!]\eta, \text{ falls } [\![e_1]\!]\eta = 0 \\ [\![e_3]\!]\eta, \text{ sonst} \end{cases} \\
[\![e_1\; \mathsf{op}\; e_2]\!]\eta & = & \begin{cases} \bot, \text{ falls } [\![e_1]\!]\eta = \bot \text{ oder } [\![e_1]\!]\eta = \bot \\ [\![e_1]\!]\eta\; \mathsf{op}\; [\![e_2]\!]\eta, \text{ sonst} \end{cases} \\
[\![\mathbf{fix}_\tau]\!]\eta & = & \lambda f{\in}[\![\tau]\!]{\Rightarrow}[\![\tau]\!].\mathsf{Y}(f)
\end{array}
$$

Satz 5.7 *Durch diese Gleichungen wird jedem Term e mit $\Gamma \vdash e : \tau$ und jeder Variablenbelegung η für Γ ein Element $[\![e]\!]\eta$ von $[\![\tau]\!]$ zugewiesen.*

Beweis. Wir haben oben schon so getan, dass durch die Gleichungen etwas definiert wird, aber das ist ja nicht unbedingt klar. Zum einen geht die Typisierung echt ein, denn die Klausel für die Applikation macht ja nur Sinn, wenn $[\![e_1]\!]\eta \in [\![\tau_2]\!] \Rightarrow [\![\tau]\!]$ und $[\![e_2]\!]\eta \in [\![\tau_2]\!]$, also das Argument der „semantischen Applikation" den erwarteten „Typ" hat. Zum anderen liefert die Klausel für die Abstraktion nur dann ein Element von $[\![\tau_1 \to \tau_2]\!]$, wenn $[\![e]\!]\eta[x{\mapsto}v]$ als Funktion von v stetig ist. Das muss man induktiv zusammen mit der Existenz der semantischen Funktion beweisen. Eingehen tut dabei, dass das Applikationsfunktional stetig ist und dass der Fixpunktoperator selbst als Funktion von $(D \Rightarrow D)$ nach D stetig ist. Die Details sind etwas langwierig und nicht weiter erhellend, sodass wir sie weglassen.

5.3 Korrektheit und Adäquatheit

Eine wichtige Eigenschaft der denotationalen Semantik ist, dass sie durch Induktion über den Termaufbau definiert ist in einer solchen Art und Weise, dass die Semantik $[\![C(e_1, e_2, \ldots, e_n)]\!]\eta$ eines zusammengesetzten Terms mit unmittelbaren Teiltermen $e_1, \ldots, e_n$ nur abhängt von der Semantik dieser Teilterme, nicht aber von den Teiltermen selbst oder von tieferliegenden Teiltermen.

Diese Eigenschaft bezeichnet man als *Kompositionalität*. Eine direkte Folgerung oder besser gesagt „mathematische Präzisierung" der Kompositionalität ist die Verträglichkeit mit der Substitution:

Satz 5.8 *Sei* $\Gamma, x{:}\,\tau_1 \vdash e : \tau$ *und* $\Gamma \vdash e_1 : \tau_1$ *und* η *eine Variablenbelegung für* Γ. *Dann gilt*

$$[\![[e_1/x]e]\!]\eta = [\![e]\!]\eta[x{\mapsto}[\![e_1]\!]\eta]$$

Beweis. Durch Induktion über den Aufbau von e.

Aus dieser Substitutivität ergibt sich unmittelbar die Korrektheit der Semantik bzgl. der Auswertung:

Satz 5.9 *Ist* e *Programm vom Typ* τ *und gilt* $e \Rightarrow e'$, *dann ist* $[\![e]\!] = [\![e']\!]$.

Beweis. Durch Induktion über den Aufbau von e, bzw. die Herleitung von $e \Rightarrow e'$. Interessant sind die Regeln E-App-B und E-Fix. Im ersten Fall haben wir

$$[\![(\mathbf{fun}(x{:}\,\tau)e)e']\!] = (\lambda v.[\![e]\!][x \mapsto v])[\![e']\!] = [\![e]\!][x \mapsto [\![e']\!]] = [\![[e'/x]e]\!]$$

Im zweiten Fall verwenden wir die Fixpunkteigenschaft:

$$[\![\mathbf{fix}_\tau\ e]\!] = \mathsf{Y}([\![e]\!]) = [\![e]\!](\mathsf{Y}([\![e]\!])) = [\![e\ (\mathbf{fix}_\tau\ e)]\!]$$

Diese Korrektheit hilft uns schon teilweise bei der Verifikation von Programmen. Haben wir etwa (wie in der Übung) gezeigt, dass die Fakultät der kleinste Fixpunkt des Fakultätsfunktional ist, so folgt mit der Korrektheit, dass $\mathsf{fact}\ n$ zumindest keinen anderen Wert als $n!$ haben kann. Es könnte aber divergieren und um das auszuschließen müssen wir einen noch engeren Zusammenhang zwischen operationaler und denotationaler Semantik herstellen, die Adäquatheit:

Theorem 5.10 (Adäquatheit) *Ist e ein Programm vom Typ* **int***, so divergiert e genau dann, wenn $[\![e]\!] = \bot$.*

Bevor wir das beweisen, wollen wir uns einige Folgerungen überlegen. Zunächst mal sehen wir daraus—wiederum unter Verwendung der Übung—dass $\mathsf{fact}\ n$ immer terminiert, denn würde $\mathsf{fact}\ n$ divergieren, so würde der kleinste Fixpunkt des Fakultätsfunktionals angewendet auf n als Ergebnis $\bot$ liefern, was ja nicht der Fall ist.

Eine etwas allgemeinere Folgerung ist folgendes Beweisprinzip für Beobachtungsgleichheiten:

Satz 5.11 *Seien e_1, e_2 Programme eines beliebigen Typs τ. Wenn $[\![e_1]\!] = [\![e_2]\!]$, dann ist $e_1 =^{\mathrm{obs}}_{\tau} e_2$.*

Beweis. Sei $f : \tau \to \mathbf{int}$ ein Programmkontext und gelte $f\ e_1 \Rightarrow^* n$. Es ist dann $n = [\![f\ e_1]\!] = [\![f]\!]([\![e_1]\!]) = [\![f]\!]([\![e_2]\!]) = [\![f\ e_2]\!]$, also muss auch gelten $f\ e_2 \Rightarrow^* n$. Divergieren kann es nämlich nicht wegen der Adäquatheit und ein anderer Wert ist schon wegen der Korrektheit ausgeschlossen.

Leider gilt die Umkehrung dieser Aussage nicht, d.h., es gibt beobachtungsgleiche Programme, die dennoch verschiedene denotationale Semantik haben. Welche das sind, werden wir später noch sehen. Halten wir für den Moment fest, dass dieser Defekt erstens bedeutet, dass die denotationale Semantik ein zwar mächtiges, aber eben doch kein vollständiges Beweisprinzip für Beobachtungsgleichheit darstellt und zweitens dass wir das Milner'sche Kontextlemma nicht unmittelbar aus der Kompositionalität der Semantik folgern können. Glücklicherweise werden wir aber den Beweis der Adäquatheit so „aufbohren" können, dass wir das Kontextlemma auch noch mit erhalten.

5.4 Die logische Relation

Kommen wir nun zum Beweis der Adäquatheit. Man möchte das durch Induktion über e zeigen, kann aber dann nicht einfach die zu beweisende Eigenschaft als Induktionshypothese nehmen, da die ja nur über Programme vom Typ **int** spricht. Verallgemeinert man die Adäquatheit naiv auf höhere Typen, dann geht die Induktion nicht durch. Die Verallgemeinerung, die wirklich funktioniert ist eine Instanz eines häufig verwendeten Prinzips, das man als *logische Relation* bezeichnet.

Wir definieren für jeden Typen τ eine Relation R^τ zwischen Programmen des Typs τ und der Menge $[\![\tau]\!]$. Die Definition erfolgt durch Induktion über den Aufbau von τ mit folgenden Klauseln.

$$\begin{array}{lcl} eR^{\mathbf{int}}d & \iff & d = \bot \vee e \Rightarrow^* d \\ eR^{\tau_1 \to \tau_2}d & \iff & \forall e_1{:}\,\tau_1.\forall d_1{\in}[\![\tau_1]\!].e_1R^{\tau_1}d_1 \Rightarrow e\ e_1R^{\tau_2}d(d_1) \end{array}$$

Man sagt, die Relation $R^{\mathbf{int}}$ wird durch die zweite Klausel zu einer logischen Relation fortgesetzt.

Unsere Strategie besteht nun darin, für jedes Programm $e : \tau$ zu zeigen $eR^\tau[\![e]\!]$. Die Adäquatheit ergibt sich dann durch Spezialisierung auf $\tau = \mathbf{int}$. Dazu müssen wir aber erst einige Eigenschaften von R^τ sammeln.

Lemma 5.12 *Sei e Programm vom Typ* τ*.*

1. *Es gilt* $e\ R^\tau\ \bot$*.*

2. *Ist* $d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \dots$ *aufsteigende Kette in* $[\![\tau]\!]$*, so folgt aus* $eR^\tau d_i$ *für alle* i*, dass auch* $eR^\tau \bigsqcup_{i\geq 0} d_i$*.*

Beweis. Durch Induktion über τ. Details als Übung.

Lemma 5.13 *Seien* e_1, e_2 *Programme eines Typs* τ *und sei* $d \in [\![\tau]\!]$*. Aus* $e_2R^\tau d$ *und* $e_1 \Rightarrow e_2$ *folgt* $e_1R^\tau d$*.*

Beweis. Durch Induktion über den Aufbau von τ. Für $\tau = \mathbf{int}$ folgt das unmittelbar aus der Definition: Entweder gilt $e_2R^{\mathbf{int}}d$ auf Grund von $d = \bot$; dann gilt natürlich auch $e_1R^{\mathbf{int}}d$. Anderenfalls haben wir $e_2 \Rightarrow^* d$ und wegen $e_1 \Rightarrow e_2$ auch $e_1 \Rightarrow^* d$, also $e_1R^{\mathbf{int}}d$.

Für $\tau = \tau_1 \to \tau_2$ verwenden wir, dass mit E-App-L aus $e_1 \Rightarrow^* e_2$ folgt $e_1e' \Rightarrow^* e_2e'$, sowie die Induktionshypothese (für $\tau := \tau_2$).

Nun können wir den Hauptsatz für logische Relationen formulieren und beweisen (Statman, Plotkin).

Theorem 5.14 *Sei* $\Gamma = x_1{:}\,\tau_1, \dots, x_n{:}\,\tau_n$ *ein Kontext, die* x_i *paarweise verschieden, sei weiter* $\Gamma \vdash e : \tau$ *ein (wohltypisierter) PCF Term,* η *eine Variablenbelegung für* Γ *und* $e_1, \dots, e_n$ *ein* n*-Tupel von geschlossenen Termen mit* $e_i : \tau_i$ *und* $e_iR^{\tau_i}\eta(x_i)$ *für* $i = 1, \dots, n$*. Dann gilt*

$$[e_n/x_n][e_{n-1}/x_{n-1}]\dots[e_1/x_1]e\ R^\tau[\![e]\!]\eta$$

Beweis. Durch Induktion über die Herleitung von $\Gamma \vdash e : \tau$. Zur Vereinfachung der Notation führen wir eine Vektorschreibweise ein: $[\vec{e}/\vec{x}]e$ statt $[e_n/x_n][e_{n-1}/x_{n-1}]\dots[e_1/x_1]e$ und $\vec{e}R^{\vec{\tau}}\eta$ statt $e_i R^{\tau_i}\eta(x_i)$ für $i = 1, \dots, n$.

Fall $e = x_i$. Dann ist $[\vec{e}/\vec{x}]e = e_i$ und die Behauptung folgt unmittelbar aus der Voraussetzung.

Fall $e = e'e''$. Dann gibt es τ_2 mit $\Gamma \vdash e' : \tau'' \to \tau$ und $\Gamma \vdash e'' : \tau''$. Die Induktionsvoraussetzung liefert $[\vec{e}/\vec{x}]e' R^{\tau'' \to \tau} [\![e']\!]\eta$ und $[\vec{e}/\vec{x}]e'' R^{\tau''} [\![e'']\!]\eta$. Daraus folgt nun unmittelbar $[\vec{e}/\vec{x}](e'e'') R^{\tau} [\![e'e'']\!]\eta$, weil $[\![e'e'']\!]\eta = [\![e']\!]\eta\ [\![e'']\!]\eta$ und $[\vec{e}/\vec{x}]e'[\vec{e}/\vec{x}]e'' = [\vec{e}](e'e'')$ nach Definition der Semantik und der Substitution.

Fall $e = \mathbf{fun}(x{:}\,\tau')e''$. Hier haben wir $\Gamma, x{:}\,\tau' \vdash e'' : \tau''$ und $\tau = \tau' \to \tau''$. Wir haben zu zeigen

$$\forall e'{:}\,\tau'.\forall d' {\in} [\![\tau']\!]. e' R^{\tau'} d' \Rightarrow (\mathbf{fun}(x{:}\,\tau')e'')(e') R^{\tau''} [\![[\vec{e}/\vec{x}](\mathbf{fun}(x{:}\,\tau')e'')e']\!]\eta$$

Seien also $e'{:}\,\tau'$ und $d \in [\![\tau']\!]$ mit $e' R^{\tau'} d'$ vorgegeben.

Die Induktionsvoraussetzung (mit $n := n+1$, $e_{n+1} := e'$, $\eta := \eta[x {\mapsto} d']$) zusammen mit der Definition der Semantik liefert

$$[e/x][\vec{e}/\vec{x}]e''\ R^{\tau''}\ [\![e]\!]\eta(d')$$

Die Behauptung folgt nun mit Lemma 5.13.

Fall $e = \mathbf{fix}_\tau$. Sei $e : \tau \to \tau$ und $d \in [\![\tau \to \tau]\!]$ mit $eR^{\tau \to \tau} d$. Wir müssen zeigen, dass $\mathbf{fix}_\tau\ e\ R^\tau\ \mathsf{Y}(d)$. Mit Lemma 5.12 genügt es dafür zu zeigen $\mathbf{fix}_\tau\ e\ R^\tau d^n(\bot)$ für alle $n \in \mathbf{N}$. Für $n = 0$ folgt das aus dem ersten Teil von Lemma 5.12, denn das besagt ja $eR^\tau\bot$ für alle e. Der Induktionsschritt geht so. Wenn $\mathbf{fix}_\tau\ e\ R^\tau d^n(\bot)$, dann auch $e\ (\mathbf{fix}_\tau\ e) R^\tau d^{n+1}(\bot)$ nach Definition von $R^{\tau \to \tau}$ und der Annahme an e, d. Die Behauptung folgt mit Lemma 5.13, da ja $\mathbf{fix}_\tau\ e \Rightarrow e\ (\mathbf{fix}_\tau\ e)$.

Die anderen Fälle sind einfacher und gehen ähnlich.

Der Beweis von Theorem 5.10 ist nach dieser Vorarbeit eine Trivialität und verbleibt als Übung.

Interessanter ist die Übereinstimmung von operationaler Äquivalenz und Beobachtungsgleichheit. Es wurde ja schon in der Übung gezeigt, dass beobachtungsgleiche Terme operational äquivalent sind. Es bleibt also nachzuweisen, dass aus $e_1 \simeq_\tau e_2$ folgt $e_1 =^{\mathrm{obs}}_\tau e_2$. Könnten wir aus der Voraussetzung $e_1 \simeq_\tau e_2$ folgern, dass $[\![e_1]\!] = [\![e_2]\!]$, so würde die Beobachtungsgleichheit mit Hilfe von Satz 5.11 folgen. Leider gilt das jedoch nicht allgemein (Gegenbeispiel kommt später) und wir müssen daher nochmal die Feinstruktur der logischen Relation analysieren. Die entscheidende Beobachtung ist, dass sie mit der operationalen Äquivalenz verträglich ist.

Lemma 5.15 *Wenn $e_1 \simeq_\tau e_2$ und $e_1\ R^\tau\ d$, so gilt $e_2\ R^\tau\ d$.*

Beweis. Durch Induktion über den Aufbau von τ. Wenn $\tau = \mathbf{int}$, dann folgt aus $e_1 \Rightarrow^* d$, dass $e_2 \Rightarrow^* d$, also $e_2\ R^\tau\ d$. Wenn $\tau = \tau' \to \tau''$, so geben wir uns e', d' mit $e' R^{\tau_1} d'$ vor und müssen zeigen $e_2 e R^{\tau''} d(d')$. Wegen $e_1 R^\tau d$ folgt $e_1 e R^{\tau''} d(d')$. Das Gewünschte folgt dann mit der Induktionshypothese für $\tau := \tau''$ wegen $e_1 e \simeq_{\tau''} e_2$.

Theorem 5.16 (Milner's Kontextlemma) *Seien e_1, e_2 Programme eines Typs τ. Es ist $e_1 =^{\mathrm{obs}}_\tau e_2$ genau dann, wenn $e_1 \simeq_\tau e_2$.*

Beweis. Wenn $e_1 =^{\mathrm{obs}}_\tau e_2$, dann gilt sicher auch $e_1 \simeq_\tau e_2$, da die operationale Äquivalenz ja auf speziellen Beobachtungskontexten beruht. Für die umgekehrte Richtung argumentieren wir wie folgt. Gelte $e_1 \simeq_\tau e_2$ und sei $e : \tau \to \mathbf{int}$ ein Beobachtungskontext. Wir müssen zeigen $ee_1 \simeq_{\mathbf{int}} ee_2$. Aus Theorem 5.14 folgt $e_1 R^\tau [\![e_1]\!]$, also mit Lemma 5.15 auch $e_2 R^\tau [\![e_1]\!]$. Weiter gilt $e R^{\tau \to \mathbf{int}} [\![e]\!]$ mit Theorem 5.14 und daher $ee_2 R^{\mathbf{int}} [\![ee_1]\!]$, da $[\![ee_1]\!] = [\![e]\!]([\![e_2]\!])$. D.h., entweder ist $[\![ee_1]\!] = \bot$; dann divergiert ee_1 wegen Theorem 5.10, oder $ee_2 \Rightarrow^* [\![ee_1]\!]$. Also $ee_1 \simeq_{\mathbf{int}} ee_2$.

Fassen wir also zusammen. Es gibt drei verschiedene Äquivalenzrelationen auf Programmen: Die Beobachtungsgleichheit, die operationale Äquivalenz und die semantische Gleichheit. Die beiden ersten sind nach Milner's Kontextlemma äquivalent; die letzte ist nach Satz 5.11 feiner als die Beobachtungsgleichheit (und die operationale Äquivalenz). In der Tat gibt es Programme, die zwar beobachtungsgleich sind, aber nicht semantisch gleich und zwar aus dem Grund, dass eine Beobachtung, welche sie unterscheidet, nicht in der Syntax definierbar ist, aber doch in der Semantik existiert. Solche Programme wollen wir jetzt kennenlernen.

5.5 Paralleles Oder und volle Abstraktion

In der Übung hatten wir bereits zwei verschiedene Arten kennengelernt, die logische Disjunktion („oder") auf möglicherweise undefinierte Werte fortzusetzen. Die Funktion $\mathsf{OR}_1 : \mathbf{int} \to \mathbf{int} \to \mathbf{int}$, die zunächst beide Argumente auswertet ud dann über den Wahrheitswert entscheidet, definiert man in PCF durch

$$\mathsf{OR}_1 \equiv \lambda x{:}\,\mathbf{int}.\lambda y{:}\,\mathbf{int}.\mathbf{sg}(x \times y)$$

wobei

$$\mathsf{sg} \equiv \lambda x{:}\,\mathbf{int}.\mathbf{if}\ x = 0\ \mathbf{then}\ 0\ \mathbf{else}\ 1$$

Daneben haben wir die Funktion, die zunächst ihr erstes Argument auswertet und nur, wenn dieses von Null verschieden (also *false*) ist, auch das zweite Argument betrachtet. Semantisch haben wir also zwei Funktionen $\mathsf{OR}_1, \mathsf{OR}_2 : \mathbf{N}_\perp \Rightarrow \mathbf{N}_\perp \Rightarrow \mathbf{N}_\perp$, die durch die Wertetabellen

OR_1	$\perp$	**true**	**false**
$\perp$	$\perp$	$\perp$	$\perp$
true	$\perp$	**true**	**true**
false	$\perp$	**true**	**false**

OR_2	$\perp$	**true**	**false**
$\perp$	$\perp$	$\perp$	$\perp$
true	**true**	**true**	**true**
false	$\perp$	**true**	**false**

Hier ist **true** $= 0$ und **false** $= 1$. Neben der zu OR_2 symmetrischen Version, die ihr zweites Argument zuerst auswertet gibt es nun noch eine vierte *parallele* Variante, die beide Argumente gleichzeitig auswertet und sobald eines der beiden den Wert **true** liefert, die andere Berechnung abbricht und als Endresultat **true** ausgibt. Liefert die schnellere Berechnung **false**, so muss natürlich auch noch das Ergebnis der zweiten abgewartet werden.

Definition 5.17 *Die Funktion* $\mathsf{POR} : \mathbf{N}_\perp \Rightarrow \mathbf{N}_\perp \Rightarrow \mathbf{N}_\perp$ *ist durch folgende Wertetabelle erklärt.*

POR	$\perp$	0	≥ 0
$\perp$	$\perp$	0	$\perp$
0	0	0	0
≥ 0	$\perp$	0	1

Obwohl POR nun wie oben begründet, intuitiv berechenbar ist, existiert in PCF kein Term des Typs **int** $\to$ **int** $\to$ **int**, dessen Semantik gleich POR wäre. Der Grund ist intuitiv das Fehlen paralleler Sprachkonstrukte in PCF. Im folgenden wollen wir beweisen, dass POR tatsächlich nicht definierbar ist. Dazu verwenden wir wieder die Methode der logischen Relation und zwar definieren wir diesmal eine Familie von Relationen $R^\tau \subseteq [\![\tau]\!]^3$ durch

$$
\begin{aligned}
&R^{\mathbf{int}}(x, y, z) \iff (x = y = z) \vee (y = \perp \wedge z = \perp) \vee (x = \perp \wedge z = \perp) \\
&R^{\tau' \to \tau''}(x, y, z) \iff \forall x', y', z' \in [\![\tau']\!]. R^{\tau'}(x', y', z') \Rightarrow R^{\tau''}(x(x'), y(y'), z(z'))
\end{aligned}
$$

$R^{\mathbf{int}}(x, y, z)$ bedeutet, dass z das Infimum von x und y in $\mathbf{N}_\perp$ ist. Man kann sich leicht überlegen, dass das auch allgemein für R^τ gilt.

Genauso, wie im vorigen Abschnitt lässt sich nun zeigen, dass die Semantik jedes Programms zu sich selbst in Relation steht:

Satz 5.18 *Wenn* $e : \tau$*, so gilt* $R^\tau([\![e]\!], [\![e]\!], [\![e]\!])$.

Beweis. Man zeigt zunächst durch Induktion über Typen, dass die R^τ unter komponentenweisen Suprema abgeschlossen sind und jeweils das Tripel $(\perp, \perp, \perp)$ enthalten. Die eigentliche Aussage ergibt sich dann als Spezialfall einer geeigneten Verallgemeinerung auf Terme mit freien Variablen, welche man durch Induktion über Typisierungsherleitungen beweist.

Korollar 5.19 *Es existiert kein Programm* $e : \mathbf{int} \to \mathbf{int} \to \mathbf{int}$*, sodass* $[\![e]\!] = \mathsf{POR}$.

Beweis. Wäre e solch ein Programm, dann wäre $([\![e]\!], [\![e]\!], [\![e]\!]) \in R^{\mathbf{int}\to\mathbf{int}\to\mathbf{int}}$. Nun ist aber $R^{\mathbf{int}}(0, \perp, \perp)$ und $R^{\mathbf{int}}(\perp, 0, \perp)$, also müsste dann auch

$$R^{\mathbf{int}}([\![e]\!](0, \perp), [\![e]\!](\perp, 0), [\![e]\!](\perp, \perp)$$

gelten. Das ist aber offensichtlich nicht der Fall, da aufgrund der Annahme an e letzteres Tripel gleich $(0, 0, \perp)$ ist.

Diese Unvollständigkeit von PCF erlaubt es uns, zwei Programme des Typs $(\mathbf{int} \to \mathbf{int} \to \mathbf{int}) \to \mathbf{int}$ zu konstruieren, die zwar beobachtungsgleich, aber nicht semantisch gleich sind. Diese Programme unterscheiden sich nämlich nur bezüglich ihrer Anwendung auf das parallele oder, was ja nicht definierbar ist.

Theorem 5.20 *Sei* $\mathsf{PORTEST} : (\mathbf{int} \to \mathbf{int} \to \mathbf{int}) \to \mathbf{int}$ *folgendes Programm:*

$$\begin{array}{l}
\mathsf{PORTEST} \equiv \mathbf{fun}(f\colon \mathbf{int} \to \mathbf{int} \to \mathbf{int}) \\
\quad\quad \mathbf{if}\ f\ 0\ \Omega \\
\quad\quad\quad \mathbf{then\,if}\ f\ \Omega\ 0 \\
\quad\quad\quad\quad \mathbf{then\,if}\ f\ 1\ 1\,\mathbf{then}\,\Omega\,\mathbf{else}\,0 \\
\quad\quad\quad\quad \mathbf{else}\,\Omega \\
\quad\quad\quad \mathbf{else}\,\Omega
\end{array}$$

Es gilt

$$\mathsf{PORTEST} \simeq^{(\mathbf{int}\to\mathbf{int}\to\mathbf{int})\to\mathbf{int}} \mathbf{fun}(f\colon(\mathbf{int} \to \mathbf{int} \to \mathbf{int}) \to \mathbf{int})\Omega$$

aber dennoch $[\![\mathsf{PORTEST}]\!] \neq \perp (= [\![\mathbf{fun}(f\colon(\mathbf{int} \to \mathbf{int} \to \mathbf{int}) \to \mathbf{int})\Omega]\!])$.

Beweis. Sei $e : \mathbf{int} \to \mathbf{int} \to \mathbf{int}$. Wir müssen zeigen, dass $\mathsf{PORTEST}e$ divergiert.Wenn nun $\mathsf{PORTEST}e$ terminiert, dann müssen auf jeden Fall die Programme $e\ 0\ \Omega$ und $e\ \Omega\ 0$ beide mit Resultat 0 terminieren. Mit Satz 5.18 wissen wir, dass $R^{\mathbf{int}\to\mathbf{int}\to\mathbf{int}}([\![e]\!], [\![e]\!], [\![e]\!])$. Das bedeutet aber, dass auch $e\ \Omega\ \Omega$ mit Resultat Null terminiert, wegen der Monotonie also e konstant ist. Es folgt, dass $e\ 1\ 1$ auch mit Resultat Null terminiert und somit $\mathsf{PORTEST}\ e$ doch divergiert.

Andererseits ist $[\![\mathsf{PORTEST}]\!](\mathsf{POR}) = 0$ und deshalb $[\![\mathsf{PORTEST}]\!] \neq \perp$.

Wir haben also gesehen, dass in PCF Beobachtungsgleichheit und semantische Gleichheit bezüglich der Domainsemantik nicht übereinstimmen. Aus diesem Grund ist das in Satz 5.11 formulierte Beweisprinzip für Beobachtungsgleicheit nicht vollständig.

Man kann dies als einen Defekt der denotationellen Semantik betrachten und versuchen, eine feinere Semantik zu finden, für welche die beiden Gleichheiten zusammenfallen. Solche Semantiken wurden erst in allerletzter Zeit gefunden, und zwar unabhängig voneinander von Abramsky, Jagadeesan, sowie von O'Hearn und Riecke. Beide Semantiken sind jedoch recht kompliziert und für praktische Programmverifikation wahrscheinlich ungeeignet.

Eine andere Möglichkeit besteht darin, auf semantische Methoden bei der Verifikation weitgehend zu verzichten und stattdessen sich auf Beweisregeln wie Milner's Kontextlemma abzustützen. Die Semantik ist dann nach wie vor als Hilfsmittel bei der Herleitung solcher Beweismethoden nützlich.

Schließlich kann man auch argumentieren, dass POR oder noch mächtigere parallele Konstrukte zur Sprache PCF hinzugenommen werden sollen. Zum Beispiel kann man PCF um eine Konstante $\mathbf{POR} : \mathbf{int} \to \mathbf{int} \to \mathbf{int}$ erweitern und die folgenden Auswerteregeln einführen:

$$\text{(E-POR-L)} \quad \frac{e_1 \Rightarrow e_1'}{\mathbf{POR}\ e_1\ e_2 \Rightarrow \mathbf{POR}\ e_1'\ e_2}$$

$$\text{(E-POR-R)} \quad \frac{e_2 \Rightarrow e_2'}{\mathbf{POR}\ e_1\ e_2 \Rightarrow \mathbf{POR}\ e_1\ e_2'}$$

$$\text{(E-POR-T-L)} \quad \frac{}{\mathbf{POR}\ 0\ e_2 \Rightarrow 0}$$

$$\text{(E-POR-T-R)} \quad \frac{}{\mathbf{POR}\ e_1\ 0 \Rightarrow 0}$$

$$\text{(E-POR-F)} \quad \frac{}{\mathbf{POR}\ n_1\ n_2 \Rightarrow 1} \quad n_1 \in \mathbf{N} \setminus \{0\}, n_2 \in \mathbf{N} \setminus \{0\}$$

Man beachte, dass diese Reduktionsregeln nichtdeterministisch sind: $\mathbf{POR}\ \Omega_{\mathbf{int}}\ 0$ *kann* zu 0 reduziert werden (mit einmaliger Anwendung von E-POR-T-R), man kann sich aber auch dumm stellen und ständig nur E-POR-L verwenden.

Die Semantik von PCF erweitern wir nun auf das parallele Oder durch die Setzung $[\![\mathbf{POR}]\!] = \mathsf{POR}$.

Es ist unmittelbar klar, dass sich die Korrektheit (Satz 5.9) verallgemeinert; das bedeutet wiederum, dass der Nichtdeterminismus der Auswerteregeln sich nur auf das Terminationsverhalten auswirkt: Wenn $e : \mathbf{int}$ und $e \Rightarrow^* v_1$ und $e \Rightarrow^* v_2$, so gilt $[\![v_1]\!] = [\![v_2]\!] = [\![e]\!]$, also $v_1 = v_2$.

Die Adäquatheit, sowie Milner's Kontextlemma gelten auch entsprechend fort. Wir müssen dazu nur einen neuen Fall im Beweis von Theorem 5.14 berücksichtigen.

Lemma 5.21 *Es gilt* **POR** $R^{\mathbf{int}\to\mathbf{int}\to\mathbf{int}}$ POR*.*

Beweis. Gelte $e_1 R^{\mathbf{int}} d_1$ und $e_2 R^{\mathbf{int}} d_2$. Wenn $\mathsf{POR}(d_1, d_2) \neq \bot$, so muss entweder d_1 oder d_2 ungleich $\bot$ sein. Wenn z.B. $d_1 = 0$, dann gilt nach Voraussetzung $e_1 \Rightarrow^* 0$, also auch **POR** e_1 $e_2 \Rightarrow^* 0$ mit E-POR-L und E-POR-T-L, Wenn $d_1 = 1$, dann muss auch $d_2 \neq \bot$ sein, z.B. $d_2 = 0$. Dann auch $e_2 \Rightarrow^* 0$ und **POR** e_1 $e_2 \Rightarrow^* 1$. Die anderen Fälle gehen ähnlich.

Was man sich bei diesem Beweis merken sollte ist, dass man die Reduktionsregeln für **POR** hätte finden können, indem man einfach versucht, den obigen Beweis durchzuziehen. Die logische Relation suggeriert also geeignete Reduktionsregeln für neue Konstrukte.

Für diese erweiterte Sprache fallen nun semantische Gleichheit und Beobachtungsgleichheit zusammen; man sagt, die Semantik ist „voll abstrakt". Um das zu zeigen benötigen wir ein Lemma, was hier nur motiviert, nicht aber vollständig bewiesen werden kann.

Lemma 5.22 *Sei τ ein Typ und $d \in [\![\tau]\!]$. Es existiert eine Folge $e_1, e_2, \ldots$ von Programmen des Typs τ, sodass die Kette*

$$[\![e_1]\!] \sqsubseteq [\![e_2]\!] \sqsubseteq \ldots$$

wie angedeutet aufsteigend ist und Supremum d besitzt.

Die grundlegende Beobachtung beim Beweis dieser Eigenschaft ist, dass jede Funktion (sagen wir in $[\![\mathbf{int} \to \mathbf{int} \to \mathbf{int}]\!]$), gleich ob berechenbar oder nicht, sich als Supremum von Funktionen mit endlichem Definitionsbereich schreiben lässt und dass solche endlichen Funktionen immer durch Terme definiert werden können (unter Verwendung einer großen Fallunterscheidung und ggf. POR). Die Schwierigkeit liegt nun darin, dies in geeigneter Weise auf höhere Typen, wie z.B. $(\mathbf{int} \to \mathbf{int}) \to \mathbf{int}$ zu verallgemeinern.

Theorem 5.23 (Volle Abstraktion) *Seien $e, e' : \tau$ Programme eines gemeinsamen Typs τ. Wenn $e =^{\mathrm{obs}}_{\tau} M'$, dann auch $[\![e]\!] = [\![e']\!]$.*

Beweis. Durch Induktion über τ. Für $\tau = \mathbf{int}$ folgt das aus der Adäquatheit. Sei also $\tau = \tau_1 \to \tau_2$. Da $[\![e_1]\!]$ und $[\![e_2]\!]$ Funktionen sind, gilt $[\![e]\!] = [\![e']\!]$, wenn für alle $d \in [\![\tau_1]\!]$ gilt $[\![e]\!](d) = [\![e']\!](d)$. Sei $d \in [\![\tau_1]\!]$ vorgegeben. Nach Lemma 5.22 gibt es eine Folge von Termen $(e_i)_i$, sodass $d = \bigsqcup_{i\geq 0} [\![e_i]\!]$. Aus $e =^{\mathrm{obs}}_{\tau} e'$ folgt $ee_i =^{\mathrm{obs}}_{\tau_2} e'e_i$, also nach Induktionshypothese $[\![e]\!][\![e_i]\!] = [\![ee_i]\!] = [\![e'e_i]\!] = [\![e']\!][\![e_i]\!]$. Aufgrund der Stetigkeit von $[\![e]\!]$ gilt dann auch $[\![e]\!](\bigsqcup_{i\geq 0}) = [\![e']\!](\bigsqcup_{i\geq 0})$, also $[\![e]\!](d) = [\![e']\!](d)$.

Dieser Beweis war völlig unabhängig von der speziellen Natur des paralleln Oder. Der eigentliche Grund, warum das Domainmodell voll abstrakt für PCF mit POR ist, liegt in der Definierbarkeitsaussage Lemma 5.22.