Erweiterung der Dynamischen Logik um temporallogische Operatoren

Steffen Schlager

19. Oktober 2000

Studienarbeit

Universität Karlsruhe (TH)
Fakultät für Informatik
Institut für Logik, Komplexität und Deduktionssysteme

Verantwortlicher Betreuer: Prof. Dr. Peter H. Schmitt
Betreuer: Dr. Bernhard Beckert

Danksagung

An dieser Stelle möchte ich mich bei meinen Betreuern Prof. Peter H. Schmitt und Dr. Bernhard Beckert bedanken, die mich bei der Erstellung dieser Studienarbeit sehr unterstützt haben. Ganz besonders bedanke ich mich bei Bernhard Beckert für das sehr interessante und ergiebige Thema und dafür, daß er sich immer Zeit für mich nahm, was oft zu ausgedehnten Diskussionen führte.

Bedanken möchte ich mich auch bei Wolfgang Ahrendt, der ebenfalls immer gerne bereit war, meine Fragen zu beantworten.

Hiermit versichere ich, die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben.

Karlsruhe, den 19. Oktober 2000

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Aufgabenstellung

Diese Studienarbeit entstand im Rahmen des KeY-Projekt [ABBG$^+$00]. Ziel des KeY-Projektes ist es, formale Methoden in den industriellen Softwareentwicklungsprozeß zu integrieren. Der Ausgangspunkt ist ein kommerzielles CASE-Tool, in das Werkzeuge zur formalen Spezifikation und Verifikation integriert werden sollen. Das Ziel dabei ist letztendlich, daß auch Programmierer, die keine oder nur wenig Erfahrung in der Anwendung formaler Methoden haben, von deren Vorteilen profitieren können. Dazu ist es notwendig, den Verifikationsprozeß so weit wie möglich vor dem Anwender zu verbergen, also die Verifikation zu automatisieren.
Die Logik, die im KeY-Projekt zur Verifikation verwendet wird, ist die Dynamische Logik [Hare84]. Ziel dieser Studienarbeit war es, die Dynamische Logik dahingehend zu erweitern, daß auch Aussagen über gewisse Zustände während des Programmablaufs möglich sind, was mit der „normalen“ Dynamischen Logik nicht ausdrückbar ist. Es hat sich herausgestellt, daß man mit dieser Erweiterung schließlich zur Prozeßlogik gelangt, über die es schon einige Arbeiten gibt [HaKo82, Prat79]. Allerdings beziehen sich die Axiomatisierungen der Prozeßlogik in diesen Arbeiten alle auf die aussagenlogische Prozeßlogik, was für den praktischen Einsatz nicht ausreicht. So ist schließlich das Ziel dieser Studienarbeit, diese Lücke zu füllen, d.h., eine Axiomatisierung der Prozeßlogik erster Stufe für eine einfache Programmiersprache anzugeben.
Nach einer kurzen Einleitung in Abschnitt 2, die einen knappen Überblick über das Gebiet der Programmverifikation gibt, wird in Abschnitt 3 das Prinzip des Sequenzenkalküls rekapituliert. In Abschnitt 4 wird die Dynamische Logik, die als Grundlage dieser Arbeit anzusehen ist, aufbereitet. Die Programmiersprache, die später für die Ableitungsregeln des Sequenzenkalküls verwendet wird, ist in Abschnitt 5 definiert. In Abschnitt 6 wird zuerst die Prozeßaussagenlogik wiederholt. Danach wird analog zum Übergang Dynamische Aussagenlogik/Dynamische Quantorenlogik eine Prozeßlogik erster Stufe definiert, für die in Abschnitt 7.2 dann ein Sequenzenkalkül aufgebaut wird, dessen Korrektheit und relative Vollständigkeit bewiesen werden. Den Abschluß der Arbeit bilden zwei Beispiele und eine kurze Diskussion.

2 Einleitung

Jeder, der schon einmal programmiert hat, wird folgende Erfahrungen gemacht haben:

- Programme enthalten Fehler.
- Das Finden und Beseitigen der Fehler dauert meistens länger als das Programmieren an sich.
- Es gelingt nicht immer, alle Fehler zu entdecken und zu beseitigen.

Die wohl am häufigsten eingesetzte Methode, um Fehler aufzuspüren, ist das Testen. Damit entdeckt man allerdings nur die Fehler, die tatsächlich wärend des Testens auftreten. Aber kein noch so intensives Testen kann garantieren, daß das Programm gänzlich fehlerfrei ist. Mit Hilfe von Tests ist es nur möglich, zu zeigen, daß Fehler enthalten sind. Wenn man aber garantieren will, daß ein Programm fehlerfrei ist, reicht Testen nicht aus. Dazu bedarf es Methoden aus dem Bereich der Logik und der Mathematik. Das Ziel ist dabei, die Korrektheit eines Programms bzw. Algorithmus formal zu beweisen. Das Charakteristische eines Beweises ist, daß die Gültigkeit einer Aussage allgemein nachgewiesen wird. Wenn es gelingt zu beweisen, daß ein Programm korrekt ist, dann bedeutet dies, daß das Programm unabhängig von der Eingabe tatsächlich immer das tut, was in der Spezifikation definiert wurde. Das ist genau das, was man durch Testen niemals erreichen kann.
Bevor allerdings genauer auf Korrektheit eingegangen werden kann, ist es notwendig, den Begriff „Korrektheit“ in Bezug auf Programmverifikation genau zu definieren. Ist ein Programm bereits korrekt, wenn der Compiler es fehlerfrei übersetzt? Ist ein Programm korrekt, wenn es nicht abstürzt, d.h. terminiert?
Unter der Korrektheit eines Programmes versteht man die Erfüllung der Spezifikation. Um also überhaupt von Korrektheit reden zu können, bedarf es einer Spezifikation. Ist aber bereits die Spezifikation fehlerhaft, dann kann es passieren, daß das Programm zwar nach der Definition des Begriffs korrekt ist, aber trotzdem nicht das tut, was man eigentlich erwartet hätte. Das ist aber ein anderes, wenn auch gewichtiges Problem, auf das hier nicht eingegangen wird. In dieser Arbeit wird die Korrektheit der Spezifikation vorausgesetzt.
Ferner wird noch zwischen partieller und totaler Korrektheit unterschieden. Eine etwas saloppe Definition, bei der aber die Bedeutung der Begriffe klar werden sollte, ist die folgende:

- Partielle Korrektheit bedeutet, daß, falls man ein Ergebnis erhält, dieses auch richtig ist. Während der Programmausführung weiß man allerdings nicht, ob das Programm sich vielleicht in einer Endlosschleife befindet oder ob noch eine Chance besteht, daß das Programm terminiert und man ein (richtiges) Ergebnis erhält.

- Totale Korrektheit gibt zusätzlich zur partiellen Korrektheit noch die Garantie der Terminierung. Man weiß also, daß man nach endlicher Zeit ein richtiges Ergebnis erhält.

Die ersten Arbeiten über Programmverifikation sind Mitte der 60er Jahre erschienen ([Thie66] und unabhängig davon [Enge67]). In den darauffolgenden Jahren betätigten sich etliche Forscher auf diesem Gebiet. Zu erwähnen sind hier besonders [Floy67] und [Hoar67]. Die Dynamische Logik ist eine relativ späte Entwicklung und wurde 1977 von Pratt [Prat77] eingeführt.
Die Dynamische Logik ist eine Erweiterung der klassischen Prädikatenlogik und greift Ideen der Modallogik[1] auf. Die ersten, wenn auch informellen Ansätze in Richtung Modallogik sind schon sehr alt und stammen aus der Antike. Der Hauptunterschied zwischen Dynamischer Logik oder Modallogik einerseits und klassischer Prädikatenlogik andererseits ist, daß sich bei ersteren der Wahrheitsgehalt einer Aussage ändern kann. In der Prädikatenlogik wird die Wahrheit einer Aussage durch die Interpretation von freien Variablen, Termen und Prädikaten bestimmt und ist danach fest. In der Dynamischen Logik hingegen kann sich der Wahrheitsgehalt einer Aussage dadurch ändern, daß Variablenbelegungen durch explizite, syntaktische Konstrukte verändert werden. Diese Konstrukte sind die einzelnen Befehle eines Programms. Wichtig zu erwähnen ist, daß diese Befehle, mit denen man die Interpretation einer Formel beeinflussen kann, Teil der Syntax sind.
In Dynamischer Logik ist es möglich, gleichzeitig das Programm als auch gewisse Eigenschaften und Anforderungen an das Programm syntaktisch auszudrücken.
Allerdings ist es nur möglich, Aussagen über das Eingabe-/Ausgabeverhalten von Programmen zu treffen, d.h., man kann nur Zusicherungen der Form

„**Nach** Ausführung des Programms gelten gewisse Eigenschaften.“

machen. Dies ist für viele Anwendungen ausreichend. Manchmal steht man aber vor dem Problem, daß ein Programm gar nicht terminiert (zum Beispiel ein Betriebssystem), man aber trotzdem beweisen möchte, daß gewisse Bedingungen während der Laufzeit eingehalten werden. Dafür ist die Dynamische Logik nicht ausdrucksstark genug.
Eine Logik, die solche Aussagen erlaubt, ist die Prozeßlogik. Die Prozeßlogik erweitert die Dynamische Logik dahingehend, daß nicht nur Eingabe-/Ausgaberelationen betrachtet werden, sondern Folgen von Zuständen, beginnend mit dem Anfangszustand und endend mit dem Terminierungszustand (falls vorhanden). Es sind auch

[1] Eine Einführung in die Modallogik geben zum Beispiel [Lemm77, Chel80, BuSe84].

unendliche Folgen von Zuständen möglich. Über diese Folgen, die den Ablauf des Programmes widerspiegeln, können nun Aussagen wie

> „**Während** der Ausführung des Programms gelten gewisse Eigenschaften.“

getroffen werden. Alle Aussagen der Dynamischen Logik lassen sich auch in der Prozeßlogik formulieren, da eine Eingabe-/Ausgaberelation der Dynamischen Logik als eine Menge von Zustandsfolgen, bei denen man alle Zwischenzustände außer Acht läßt, aufgefaßt werden kann.

3 Sequenzenkalkül

Um Korrektheitsbeweise von Programmen in Dynamischer Logik automatisieren zu können, ist eine Axiomatisierung[2] erforderlich. In dieser Arbeit wird dazu der von Gentzen [Gent35] eingeführte Sequenzenkalkül verwendet, da formale Beweise in einem Sequenzenkalkül informellen von-Hand-Beweisen oft sehr nahe kommen und deshalb relativ verständlich sind.

Definition 1 (Sequenz).
Eine Sequenz besteht aus zwei endlichen Mengen von Formeln[3] *und hat folgende Gestalt:*

$$\Gamma \ \vdash \ \Delta.$$

Ist $\Gamma = \{\varphi_1, \varphi_2, \ldots, \varphi_n\}$ *und* $\Delta = \{\psi_1, \psi_2, \ldots, \psi_m\}$*, dann schreibt man auch*

$$\varphi_1, \varphi_2, \ldots, \varphi_n \ \vdash \ \psi_1, \psi_2, \ldots, \psi_m.$$

Formeln werden mit kleinen griechischen Buchstaben bezeichnet. Große griechische Buchstaben bezeichnen Mengen von Formeln. Als Abkürzung wird statt

$$\Gamma \cup \varphi \ \vdash \ \Delta$$

meistens

$$\Gamma, \ \varphi \ \vdash \ \Delta$$

geschrieben.
Die Formelmenge links bzw. rechts von $\vdash$ wird als Antezedent bzw. Sukzedent bezeichnet.
Eine Sequenz ist eigentlich nur eine übersichtliche Schreibweise, um bei einer Disjunktion von Formeln die negierten von den nichtnegierten Formeln zu trennen. So ist mit der Sequenz $\Gamma \vdash \Delta$ die Implikation $\bigwedge \Gamma \to \bigvee \Delta$ gemeint.
Eine leerer Antezedent ist definitionsgemäß wahr, ein leerer Sukzedent definitionsgemäß falsch. Eine leere Sequenz ist ebenfalls definitionsgemäß falsch.

Definition 2 (Sequenzenregel).
Seien $S, S_1, S_2, \ldots, S_n (n \geq 0)$ *Sequenzen. Dann heißt folgendes Schema Sequenzenregel:*

$$\frac{S_1 \quad S_2 \quad \ldots \quad S_n}{S}$$

[2]Unter „Axiomatisierung“ wird im folgenden immer ein Kalkül verstanden.
[3]Was eine Formel ist, wird später definiert.

S wird als Konklusion der Sequenzenregel bezeichnet, $S_1, S_2, \ldots, S_n$ als ihre Prämissen. Axiome sind Regeln ohne Prämissen, d.h. ein Axiom ist eine Sequenzenregel der Form

$$\frac{}{S}$$

Die Semantik einer Sequenzenregeln ist: sind alle Prämissen ableitbar, dann ist auch die Konklusion ableitbar.
Der Sequenzenkalkül wird normalerweise „rückwärts“ angewandt, d.h. man beginnt mit der zu beweisenden Sequenz und benutzt die Sequenzenregeln derart, daß man eine Konklusion durch die (hoffentlich einfacheren) Prämissen ersetzt. Das Ziel ist es, am Schluß nur noch Axiome übrig zu haben. Gelingt dies, dann ist der Beweis geschlossen.

Schließlich sollen noch die *Korrektheit* und die *Vollständigkeit* definiert werden. Dazu müssen vorher noch die Begriffe der *semantischen Folgerbarkeit* und der *Ableitbarkeit* eingeführt werden.[4]

Definition 3 (Semantische Folgerbarkeit).
Seien Σ eine Signatur und Γ, φ eine Formelmenge bzw. Formel über Σ. Dann definiert man:

$$\Gamma \models_\Sigma \varphi \textit{ gdw. Jedes Modell von } \Gamma \textit{ ist auch Modell von } \varphi.$$

Statt $\emptyset \models_\Sigma \varphi$ schreibt man kurz $\models_\Sigma \varphi$. Wenn klar ist, was die Signatur Σ ist, schreibt man oft $\models$ anstatt $\models_\Sigma$.

Definition 4 (Ableitbarkeit im Sequenzenkalkül).
Eine Ableitung im Sequenzenkalkül ist eine endliche Folge von Sequenzen, die mittels der Sequenzenregeln und der Axiome gebildet werden. Man sagt, φ sei aus Γ im Sequenzenkalkül ableitbar, bezeichnet mit

$$\Gamma \vdash_\Sigma \varphi,$$

gdw. eine Ableitung der Sequenz $\Gamma \vdash_\Sigma \varphi$ im Sequenzenkalkül existiert.

Die Formeln, die in der Menge Γ enthalten sind, sind als Voraussetzungen für eine Ableitung anzusehen. Will man die Allgemeingültigkeit einer Sequenz zeigen, dann ist $\Gamma = \emptyset$.

[4]Dabei taucht der Begriff „Modell“ auf, wobei die Definition der üblichen Definition aus der Prädikatenlogik entspreche.

Man beachte, daß das Symbol $\vdash$ in diesem Zusammenhang mit zwei verschiedenen Bedeutungen belegt ist. Es wird gleichzeitig zur Trennung von Antezedent und Sukzedent in Sequenzen verwendet als auch, wie gerade definiert, zur Kennzeichnung der Ableitbarkeit im Sequenzenkalkül.

Definition 5 (Korrektheit).
Sei Γ, φ eine Menge von Formeln bzw. eine Formel über der zugrunde gelegten Signatur Σ. Dann gilt:

$$\Gamma \vdash \varphi \textit{ impliziert } \Gamma \models \varphi.$$

Definition 6 (Vollständigkeit).

$$\Gamma \models \varphi \textit{ impliziert } \Gamma \vdash \varphi.$$

Korrektheit bedeutet, daß man durch Anwendung des Kalküls nur Formeln bzw. Sequenzen ableiten kann, die auch semantisch folgerbar sind. Umgekehrt bedeutet Vollständigkeit, daß alles, was semantisch ableitbar ist, auch mir Hilfe des Kalküls syntaktisch abgeleitet werden kann.

4 Dynamische Logik

Mit der Dynamischen Logik ist es möglich, bestimmte Eigenschaften von Programmen, zum Beispiel die Terminierung, zu beschreiben. Die Erfüllung dieser Eigenschaften kann mit Hilfe von Kalkülen formal bewiesen werden. Dazu ist allerdings eine geeignete Axiomatisierung der Dynamischen Logik erforderlich. Eine mögliche Axiomatisierung wird später vorgestellt.
Die Formeln der Dynamischen Logik enthalten Programme und aussagenlogische bzw. prädikatenlogische Formeln, abhängig davon, ob man die Dynamische Aussagenlogik (Abschnitt 4.1) oder die Dynamische Quantorenlogik (Abschnitt 4.2) betrachtet. Je nachdem, was man für Programmkonstrukte zuläßt, erhält man verschiedene Varianten mit verschiedener Ausdrucksstärke. Man sollte also nicht den Eindruck gewinnen, daß die im folgenden vorgestellte Dynamische Logik die einzig mögliche ist. Am Ende dieses Kapitels werden noch einige andere Varianten kurz vorgestellt.
Zuerst wird jedoch eine Dynamische Aussagenlogik, dann eine Dynamische Quantorenlogik definiert. Analog zu der Unterscheidung zwischen Dynamischer Aussagenlogik und Dynamischer Quantorenlogik ist die Unterscheidung von Aussagenlogik und Prädikatenlogik in der klassischen Logik. Konkret bedeutet das, daß sich im aussagenlogischen Fall die Menge der erlaubten Programme nur aus Verknüpfungen atomarer anonymer Programme zusammensetzt. Die atomaren Programme sind vergleichbar mit den aussagenlogischen Variablen. Da auf der aussagenlogischen Ebene keine Variablen auftauchen, ist es nicht möglich, das einfachste und wichtigste Programmkonstrukt, nämlich die Zuweisung, auszudrücken. Dies ist erst in der Dynamischen Quantorenlogik möglich. Dort werden atomare Programme durch konkrete Programmbefehle, nämlich durch Zuweisungen, ersetzt, analog zur Prädikatenlogik, wo die aussagenlogischer Variablen durch atomare Formeln ersetzt werden.

4.1 Dynamische Aussagenlogik

4.1.1 Syntax

Definition 7 (Syntax für die Dynamische Aussagenlogik).
Eine Signatur $\Sigma = \langle \Pi_0, F_0 \rangle$ *besteht aus einer nichtleeren Menge* Π_0 *atomarer Programme und einer Menge* F_0 *aussagenlogischer Variablen. Die Menge* Fml_{dal} *der Formeln der Dynamischen Aussagenlogik* Fml_{dal} *und die Menge* Π *der erlaubten Programme wird simultan induktiv definiert durch:*

- *true, false* $\in Fml_{dal}$,

- $F_0 \subseteq Fml_{dal}$,
- *sind* F_1, $F_2 \in Fml_{dal}$, *dann auch* $\neg F_1$, $(F_1 \vee F_2)$,
- $\Pi_0 \subseteq \Pi$,
- *sind* $\alpha, \beta \in \Pi$, *dann auch* $(\alpha;\beta)$, $(\alpha \cup \beta)$ *und* (α^*),
- *sind* $F \in Fml_{dal}$ *und* $\alpha \in \Pi$, *dann ist* $\langle\alpha\rangle F \in Fml_{dal}$,
- *ist* $F \in Fml_{dal}$ *und* F *programmfrei, dann ist* $(F?) \in \Pi$.

Klammern werden häufig der besseren Übersicht wegen weggelassen, falls dadurch keine Unklarheiten entstehen. Außerdem werden folgende gebräuchliche Abkürzungen definiert:

- $F_1 \wedge F_2$ für $\neg(\neg F_1 \vee \neg F_2)$,
- $F_1 \rightarrow F_2$ für $\neg F_1 \vee F_2$,
- $F_1 \leftrightarrow F_2$ für $(F_1 \rightarrow F_2) \wedge (F_2 \rightarrow F_1)$ und
- $[\alpha]F$ für $\neg\langle\alpha\rangle\neg F$.

Die Formeln $\langle\alpha\rangle F$ und $[\alpha]F$ werden gesprochen „Diamond α F“ bzw. „Box α F“. In Abbildung 2 sind einige Beispiele gültiger Formeln der Dynamischen Aussagenlogik abgebildet.

4.1.2 Semantik

Bevor im folgenden die Semantik der Dynamischen Aussagenlogik genau definiert wird, soll deren beabsichtigte Bedeutung informell veranschaulicht werden:

$\langle\alpha\rangle F$	Es gibt eine Ausführung von Programm α, so daß danach F gilt.
$[\alpha]F$	Nach jeder möglichen Ausführung von Programm α gilt F.
$(\alpha;\beta)$	Hintereinanderausführung von α und β.
$(\alpha \cup \beta)$	Nichtdeterministische Ausführung von entweder α oder β.
(α^*)	Nichtdeterministisch endlich häufige Wiederholung von α.
$(F?)$	Fortsetzung nur wenn F wahr ist, ansonsten keine Terminierung.

Programme werden in Dynamischer Logik als binäre Relationen über einer gewissen Menge W von Zuständen interpretiert. In der Dynamischen Aussagenlogik wird W nicht weiter präzisiert, d.h., man kann beliebige Mengen verwenden. Ein Tupel $(s,t) \in W \times W$ liegt genau in der Relation für das Programm $\pi \in \Pi$, wenn π, im Zustand s gestartet, im Zustand t enden kann.

Die Verwendung der Operatoren „$\cup$“ und „$*$“ hat zur Folge, daß die Programme nichtdeterministisch sind, was bei der Definition der Semantik die Verwendung von Relationen zur Interpretation von Programmen anstatt Funktionen nötig macht. Jedoch wird dieser Indeterminismus später bei der Aufstellung der Regeln für den Kalkül fallen gelassen, da die in Abschnitt 5 definierte Programmiersprache deterministisch ist. In Abbildung 1 sind einige deterministische Programmkonstrukte und deren Darstellung in Dynamischer Logik abgebildet.
Die Definition der Semantik basiert auf *Strukturen.*

Definition 8 (DAL-Struktur).
Eine Struktur $\mathcal{A} = (W, \tau, \rho)$ für die Dynamische Aussagenlogik besteht aus

- *einer nichtleeren Menge W von Zuständen,*
- *einer Interpretation $\tau : F_0 \to 2^W$ der aussagenlogischen Variablen,*
- *einer Interpretation $\rho : \Pi_0 \to W \times W$ der atomaren Programme.*

Die Funktion τ ordnet jeder aussagenlogischen Variablen $p \in F_0$ die Menge der Zustände zu, in denen p wahr ist. ρ ordnet jedem atomaren Programm $\pi \in \Pi_0$ die Paare von Zuständen zu, die bezüglich π in Relation zueinander stehen.

Definition 9 (Semantik für die Dynamische Aussagenlogik).
Sei $\mathcal{A} = (W, \tau, \rho)$ eine Struktur für die Dynamische Aussagenlogik, dann wird die Funktion τ auf die Menge aller Formeln F fortgesetzt und ρ wird auf die Menge aller Programme Π fortgesetzt durch folgende simultane induktive Definition:

- $\tau(wahr) = W$ *und* $\tau(falsch) = \emptyset$,
- $\tau(\neg F) = W \setminus \tau(F)$,
- $\tau(F_1 \vee F_2) = \tau(F_1) \cup \tau(F_2)$,
- $\tau(\langle\alpha\rangle F) = \{s \in W \mid$ *es gibt ein* $t \in \tau(F)$ *mit* $(s,t) \in \rho(\alpha)\}$,
- $\rho(\alpha;\beta) = \{(s,t) \in W \times W \mid$ *es gibt ein* $u \in W$ *mit* $(s,u) \in \rho(\alpha)$ *und* $(u,t) \in \rho(\beta)\}$,
- $\rho(\alpha \cup \beta) = \rho(\alpha) \cup \rho(\beta)$,
- $\rho(\alpha^*) = \{(s,t) \mid$ *es gibt ein* $k \geq 0$ *und es gibt* $s_0, \ldots, s_k \in W$ *mit* $s_0 = s, s_k = t$ *und* $(s_{i-1}, s_i) \in \rho(\alpha)$ *für* $1 \leq i \leq k\}$,
- $\rho(F?) = \{(s,s) \mid s \in \tau(F)\}$.

Definition 10.
Sei $\mathcal{A} = (W, \tau, \rho)$ eine Struktur der Dynamischen Aussagenlogik, $w \in W$ und $F \in Fml_{dal}$. Dann benutzt man folgende Terminologie:

w erfüllt F oder F ist wahr in w	$w \models F$	*für*	$w \in \tau(F)$,
F ist $\mathcal{A}$-gültig	$\mathcal{A} \models F$	*für*	$w \in \tau(F)$ *für alle* $w \in W$,
F ist allgemeingültig	$\models F$	*für*	F *ist $\mathcal{A}$-gültig für alle* $\mathcal{A}$.

Man sollte allerdings beachten, daß das Symbol $\models$ hier nicht die Bedeutung der Folgerbarkeit hat wie in Abschnitt 3.

4.1.3 Anmerkungen

Es ist möglich, für die Dynamische Aussagenlogik eine vollständige und korrekte Axiomatisierung anzugeben (siehe zum Beispiel [Schm99, Seite 180]). Die Axiome und Regeln werden dort als das logische System DAL bezeichnet.

Satz 11.
Das Erfüllbarkeitsproblem für DAL ist entscheidbar.

Beweis: Siehe Theorem 2.12 in [Hare84, Seite 515].

Später wird sich zeigen, daß für die Dynamische Quantorenlogik ein endlicher, korrekter und vollständiger Kalkül[5] nicht existiert.

4.1.4 Beispiele

Abbildung 1 zeigt einige gängige Programmkonstrukte und ihre Darstellung in Dynamischer Aussagenlogik.
Einige allgemeingültige Formeln der Dynamischen Aussagenlogik sind in Abbildung 2 dargestellt. Die dazugehörigen Beweise findet man zum Beispiel in [Hare84].

`IF` A `THEN` α `ELSE` β	für	$(A?;\alpha) \cup (\neg A?;\beta)$
`WHILE` A `DO` α	für	$(A?;\alpha)^*;\neg A?$
`REPEAT` α `UNTIL` A	für	$\alpha;(\neg A?;\alpha)^*;A?$

Abbildung 1: Gängige Programmkonstrukte

[5] Nach Definition ist ein Kalkül immer endlich, d.h. es sind nur Regeln mit endlich vielen Prämissen zugelassen. Weicht man diese Definition auf und läßt auch Regeln mit unendlich vielen Prämissen zu, dann spricht man von einem infinitären Kalkül.

$$\langle\alpha \cup \beta\rangle(F \wedge G) \leftrightarrow (((\langle\alpha\rangle F \wedge \langle\alpha\rangle G) \vee (\langle\beta\rangle F \wedge \langle\beta\rangle G))$$
$$([\alpha]F \wedge \langle\alpha\rangle true) \rightarrow \langle\alpha\rangle F$$
$$\langle\alpha\rangle(F \vee G) \leftrightarrow (\langle\alpha\rangle F \vee \langle\alpha\rangle G)$$
$$\langle\alpha\rangle(F \wedge G) \rightarrow (\langle\alpha\rangle F \wedge \langle\alpha\rangle G)$$

Abbildung 2: Allgemeingültige Formeln der Dynamischen Aussagenlogik

Warum bei der letzten Formel die umgekehrte Richtung der Implikation nicht allgemeingültig ist, zeigt anschaulich Abbildung 3. Wählt man für die Programmausführung von α einmal den linken und einmal den rechten Ast, dann gilt sowohl $1 \models \langle\alpha\rangle F$ als auch $1 \models \langle\alpha\rangle G$. Damit gilt auch $1 \models \langle\alpha\rangle F \wedge \langle\alpha\rangle G$. Andererseits gilt aber offensichtlich nicht $1 \models \langle\alpha\rangle(F \wedge G)$, denn betrachtet man eine Ausführung von α, dann befindet man sich danach entweder im Zustand „2“ oder im Zustand „3“. Und in beiden ist $F \wedge G$ falsch.

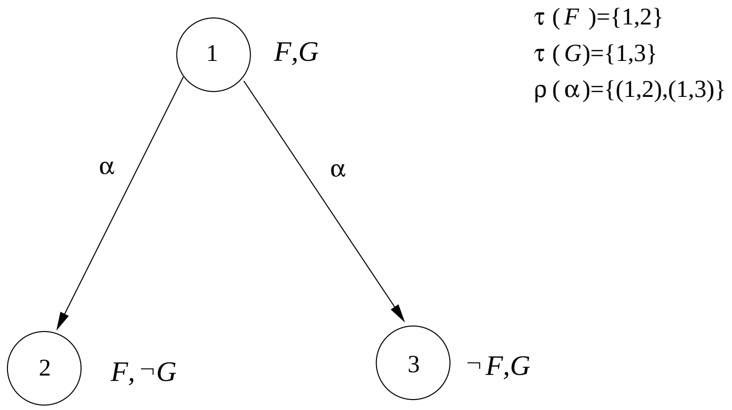

Abbildung 3: Gegenbeispiel

4.2 Dynamische Quantorenlogik

In diesem Abschnitt wird die Dynamische Quantorenlogik (englisch: First-order Dynamic Logic) definiert. Sie ist im Gegensatz zur Dynamischen Aussagenlogik dazu geeignet, Aussagen über „echte“ Programme zu treffen, die nicht mehr nur aus uninterpretierten atomaren Programmen zusammengesetzt sind. Tests ermöglichen, prädikatenlogische Formeln auf deren Gültigkeit hin abzufragen und mit Hilfe von Zuweisungen ist es möglich, die Werte von Variablen zu manipulieren. Außerdem sind programmfreie[6] Formeln nicht mehr länger auf die Aussagenlogik

[6]Programmfreie Formeln sind Formeln in denen kein Programm enthalten ist.

beschränkt, sondern es sind prädikatenlogische Formeln möglich.

4.2.1 Syntax

Tauchen bei der nun folgenden Definition der Syntax der Dynamischen Quantorenlogik und auch im weiteren Verlauf der Arbeit die Begriffe „atomare Formel“ und „Term“ auf, so seien diese definiert, wie es in der Prädikatenlogik üblich ist.

Definition 12 (Formeln und Programme).
Eine Signatur $\Sigma = \langle P, F \rangle$ *für die Dynamische Quantorenlogik besteht aus einer Menge* P *der Prädikatszeichen*[7] *und einer Menge* F *der Funktionszeichen. Die Menge der Individuenvariablen sei mit* Var *bezeichnet. Die Menge der Formeln der Dynamischen Quantorenlogik* Fml_{dql} *und der erlaubten Programme* Π *wird simultan induktiv definiert durch:*

- *true, false* $\in Fml_{dql}$,
- *jede atomare Formel ist in* Fml_{dql},
- *sind* F_1, $F_2 \in Fml_{dql}$ *und ist* x *eine Variable, dann sind auch* $\neg F_1$, $(F_1 \vee F_2)$, $\exists x F_1 \in Fml_{dql}$,
- *für jede Variable* $x \in Var$ *und jeden Term* t *ist* $(x := t) \in \Pi$,
- *für* $F \in Fml_{dql}$ *gilt* $(F?) \in \Pi$, *falls* F *programmfrei,*
- *sind* $\alpha, \beta \in \Pi$, *dann sind auch* $(\alpha; \beta)$, $(\alpha \cup \beta)$ *und* $(\alpha^*) \in \Pi$,
- *für* $F \in Fml_{dql}$ *und* $\alpha \in \Pi$ *gilt* $\langle \alpha \rangle F \in Fml_{dql}$.

Wie bei der Dynamischen Aussagenlogik werden überflüssige Klammern oft weggelassen. Ferner werden wieder die üblichen Abkürzungen $[\alpha], \wedge, \rightarrow, \leftrightarrow$ verwendet.

4.2.2 Semantik

Zur Auswertung der Formeln der Dynamischen Quantorenlogik wird eine gewöhnliche Struktur $\mathcal{M}$ der Prädikatenlogik erster Ordnung benötigt.

Definition 13 (Struktur für die Prädikatenlogik erster Stufe).
Eine Struktur $\mathcal{M} = (U_{\mathcal{M}}, I_{\mathcal{M}})$ *für die Prädikatenlogik erster Stufe besteht aus*

- *einer nichtleeren Menge* $U_{\mathcal{M}}$, *genannt Universum,*

[7] Das Gleichheitsprädikat soll immer in P enthalten sein und in der gewohnten Weise interpretiert werden.

- *einer Interpretation $I_{\mathcal{M}}$ der Funktions- und Prädikatszeichen.*

Ferner wird eine Menge $W_{\mathcal{M}}$ an Zuständen gebraucht. Ein Zustand ist nichts anderes als eine Variablenbelegung in der Prädikatenlogik erster Stufe. So entspricht die Menge der Zustände $W_{\mathcal{M}} = \{w \mid w : Var \to U_{\mathcal{M}}\}$. Das bedeutet, daß in einem bestimmten Zustand die Variablenbelegung fest ist. Sie kann sich nur durch Übergang in einen anderen Zustand ändern.
In der Literatur (zum Beispiel in [Hare79]) wird zum Teil auch die Interpretation der Funktions- und Prädikatszeichen zum Zustand hinzugenommen. Außerdem werden dort auch uninterpretierte Funktions- und Prädikatszeichen erlaubt. Schließlich beschränkt man sich aber doch auf ein sogenanntes *simple universe*, bei dem die Interpretation der Funktions- und Prädikatszeichen in allen Zuständen fest ist und bei dem nur 0-stellige Funktionssymbole, d.h. Variablen, uninterpretiert sein dürfen, was dann wieder der Definition in dieser Arbeit entspricht.
Durch die Definition, daß die Menge W der Zustände der Menge der Variablenbelegungen entspricht, kann es nicht vorkommen, daß ein Programm einen Zustand erreicht (zum Beispiel durch eine Zuweisung), der nicht gar nicht im Modell existiert.
Bei der folgenden Definition der Semantik unterscheiden sich nur die ersten beiden Punkte von der Definition der Semantik der Dynamischen Aussagenlogik.

Definition 14 (Semantik der Dynamischen Quantorenlogik).
Sei $\mathcal{M} = (U_{\mathcal{M}}, I_{\mathcal{M}})$ eine Struktur für die Prädikatenlogik erster Stufe, $W = \{w \mid w : Var \to U_{\mathcal{M}}\}$ die Menge der Zustände und $w, w_1, w_2 \in W$, dann wird die Semantik der Dynamischen Quantorenlogik simultan induktiv definiert durch (definiert werden ρ und τ):

- $w \in \tau(R(t_1, \ldots, t_n))$ *gdw. die atomare Formel* $R(t_1, \ldots, t_n)$ *ist wahr in der Struktur* $\mathcal{M}$ *im Zustand* w*, d.h.* $val_{\mathcal{M},w}(R(t_1, \ldots, t_n)) = wahr$*, wobei* val *wie in der Prädikatenlogik üblich definiert sei,*

- $\rho(x := t) = \{(w_1, w_2) \mid val_{\mathcal{M},w_2}(y) = val_{\mathcal{M},w_1}(y)$ *für* $y \neq x$ *und* $val_{\mathcal{M},w_2}(x) = val_{\mathcal{M},w_1}(t)\}$,

- $\tau(wahr) = W$ *und* $\tau(falsch) = \emptyset$,

- $\tau(\neg F) = W \setminus \tau(F)$,

- $\tau(F_1 \vee F_2) = \tau(F_1) \cup \tau(F_2)$,

- $\tau(\exists x F) = \{w \mid$ *es existiert ein* $d \in U_{\mathcal{M}}$ *mit* $w_x^d \in \tau(F)\}$*, wobei* w_x^d *die Modifikation des Zustands, d.h. der Variablenbelegung an der Stelle* x *zu* d *ist.*

- $\tau(\langle\alpha\rangle F) = \{s \in W \mid$ *es gibt ein* $t \in W$ *mit* $(s,t) \in \rho(\alpha)$ *und* $t \in \tau(F)\}$,
- $\rho(\alpha;\beta) = \{(s,t) \in W \times W \mid$ *es gibt ein* $u \in W$ *mit* $(s,u) \in \rho(\alpha)$ *und* $(u,t) \in \rho(\beta)\}$,
- $\rho(\alpha \cup \beta) = \rho(\alpha) \cup \rho(\beta)$,
- $\rho(\alpha^*) = \{(s,t) \mid$ *es gibt ein* $k \geq 0$ *und es gibt* $s_0, \ldots, s_k \in W$ *mit* $s_0 = s, s_k = t$ *und*
  $(s_{i-1}, s_i) \in \rho(\alpha)$ *für* $1 \leq i \leq k\}$,
- $\rho(F?) = \{(s,s) \mid s \in \tau(F)\}$.

Anstellte von $w \in \tau(F)$ schreibt man oft $(\mathcal{M}, w) \models F$. Letzteres wird gelesen als „die Formel F ist wahr im Zustand w in $\mathcal{M}$".
Im weiteren Verlauf der Arbeit wird für die Auswertungsfunktion anstatt $val_{\mathcal{M},w}$ auch oft $I(\mathcal{M}, w)$ geschrieben.

Definition 15.
Man sagt, F *ist* **gültig** *in* $\mathcal{M}$*, wenn* $(\mathcal{M}, w) \models F$ *für jeden Zustand* $w \in W_{\mathcal{M}}$ *gilt.*
F *heißt* **allgemeingültig**, *falls* F *in jeder Struktur* $\mathcal{M}$ *gültig ist.*

4.3 Axiomatisierung (Kalkül)

Wie schon in Abschnitt 4.1 erwähnt wurde, ist ein endlicher, vollständiger und korrekter Kalkül für die Dynamische Quantorenlogik nicht möglich. Den Beweis dafür findet man zum Beispiel in [Hare84, Seite 559–560]. Man kann sich diese Tatsache aber auch durch folgende Überlegung klarmachen: Nach dem Gödelschen Unvollständigkeitssatz gibt es keinen endlichen, vollständigen und korrekten Kalkül für die Arithmetik in der Prädikatenlogik erster Stufe. Da man aber mit Hilfe von Programmen die Arithmetik in Dynamischer Quantorenlogik simulieren kann, folgt, daß auch für die Dynamische Quantorenlogik ein solcher Kalkül nicht existieren kann.
In der Praxis ist man aber nicht an allgemeingültigen Formeln interessiert, sondern man beschränkt sich auf eine Struktur $\mathcal{M}$ und ist dann an den in $\mathcal{M}$ gültigen Formeln interessiert. Als Universum wählt man sinnvollerweise die der Programmiersprache zugrunde liegenden Datentypen, zum Beispiel die natürlichen Zahlen. Allerdings ist für die meisten Strukturen $\mathcal{M}$ bereits die Menge der in $\mathcal{M}$ gültigen Formeln der Prädiaktenlogik erster Stufe nicht mehr entscheidbar, so daß man auch trotz Beschränkung auf eine feste Struktur keinen finitären, korrekten und vollständigen Kalkül erwarten kann.
Allerdings gibt es Strukturen, sogenannte **arithmetische Strukturen**, für die

man zumindest einen infinitären, korrekten und vollständigen Kalkül angeben kann.

Definition 16 (Arithmetische Struktur).
Eine Struktur $\mathcal{M}$ heißt arithmetische Struktur, wenn das Universum $U_{\mathcal{M}}$ abzählbar ist und wenn gilt:

- $\mathbb{N} \subseteq U_{\mathcal{M}}$,
- $nat, 0, succ, +, *, <, = \quad \in \Sigma$.

Dabei sollen $0, +, *, <, =$ interpretiert werden, wie in $\mathbb{N}$ üblich. Das Prädikat $nat(x)$ soll genau dann wahr sein, wenn $x \in \mathbb{N}$.

Anmerkung: Man könnte auch Strukturen zulassen, die nur isomorph zu $\mathbb{N}$ sind.

Nun läßt sich ein infinitärer Kalkül[8] angeben, der für arithmetische Strukturen vollständig und korrekt ist. Dazu werden alle gültigen Formeln der Prädiaktenlogik erster Stufe als Axiome hinzugenommen.

Definition 17 (Das logische System $DQL_{\mathcal{M}}$).
Sei $\mathcal{M}$ eine arithmetische Struktur, T_{dal} die Menge der Tautologien der Dynamischen Aussagenlogik und G_{pk1} die Menge der in $\mathcal{M}$ gültigen Formeln der Prädikatenlogik erster Stufe. Das logische System $DQL_{\mathcal{M}}$ besteht aus den folgenden Axiomen und Regeln.

$$\frac{}{F} \quad F \in T_{dal} \cup G_{pk1} \tag{A1}$$

$$\frac{}{\langle \mathbf{x}{:=}t \rangle F \leftrightarrow F_x^t} \tag{A2}$$

$$\frac{F \quad F \to G}{G} \tag{MP}$$

$$\frac{F}{[\alpha]F} \quad und \quad \frac{F}{\forall x F} \tag{G}$$

$$\frac{\forall n (F(n+1) \to \langle \alpha \rangle F(n))}{\forall n (F(n) \to \langle \alpha^* \rangle F(0))} \quad F \text{ aus } PK1 \text{ und } n \in Var \text{ nicht in } \alpha \text{ enthalten} \tag{*}$$

Wenn F in dem System $DQL_{\mathcal{M}}$ ableitbar ist, schreibt man dafür $\vdash_{DQL_{\mathcal{M}}} F$.

[8]Der hier vorgestellte Kalkül ist ein Hilbert-Kalkül, da dafür nur wenige Regeln aufgestellt werden müssen und keine umfangreichen Beweise geführt werden sollen.

Satz 18 (Relative Vollständigkeit).
Sei $\mathcal{M}$ eine arithmetische Struktur. Dann gilt für jede Formeln $F \in Fml_{dql}$:

$$\mathcal{M} \models F \textit{ gdw. } \vdash_{DQL_{\mathcal{M}}} F.$$

Beweis: Siehe Theorem 3.19 in [Hare84, Seite 565].

Die speziellen Eigenschaften einer arithmetischen Struktur werden insbesondere bei der Regel (*) ausgenutzt, um sicherzustellen, daß Schleifen terminieren. Man könnte die Regeln auch anders formulieren, so daß man n auch um mehr als Eins verringern darf:

$$\frac{(n>0 \wedge F(n)) \rightarrow \langle\alpha\rangle \exists m(m<n \wedge F(m))}{F(n) \rightarrow \langle\alpha^*\rangle F(0)} \quad (*')$$

Wichtig ist nur, daß n monoton gegen 0 strebt.

4.4 Beispiele

Als Beispiel soll ein Algorithmus[9] zur Berechnung der Fakultät (Abbildung 4) dienen. Es soll gezeigt werden, daß dieser Algorithmus für nichtnegative Eingaben die

```
x:=n
y:=1
WHILE x>0 DO
   y:=y*x
   x:=x-1
```

Abbildung 4: Algorithmus zur Berechnung von `n!`

Fakultät der Eingabe berechnet und insbesondere terminiert. Um dies zu zeigen, muß man folgende, bereits etwas umgeformte Formel, die den Algorithmus und seine Spezifikation in Dynamischer Quantorenlogik ausdrückt, im logischen System $DQL_{\mathcal{N}}$ [10] ableiten.

$$\underbrace{(x = n \wedge x \geq 0 \wedge y = 1)}_{F} \rightarrow \langle(\underbrace{x > 0?; y := y * x; x := x - 1}_{\alpha})^*\rangle \underbrace{(x = 0 \wedge y = n!)}_{G}$$

[9] Der Algorithmus ist in der Programmiersprache notiert, die in Kapitel 5 definiert ist.

[10] Als Struktur wurde, wie nicht anders zu erwarten, $\mathcal{N} = (\mathbb{N}, +, \times)$ gewählt, d.h. die natürlichen Zahlen mit den üblichen Verknüpfungen.

Diese Formel wird durch $F \to \langle\alpha^*\rangle G$ abgekürzt. Als nächstes wird die Formel $H(m)$ definiert durch $H(m) = (n! = x! * y) \wedge (m = x)$. $H(m)$ stellt eine Art Induktionsannahme dar.
Weiterhin sind folgende drei Ableitungen möglich:

(i) $\vdash_{DQL_{\mathcal{N}}} F \to \exists m H(m)$
(ii) $\vdash_{DQL_{\mathcal{N}}} H(0) \to G$
(iii) $\vdash_{DQL_{\mathcal{N}}} H(m+1) \to \langle\alpha\rangle H(m)$

Aus (iii) folgt mit (*) $\vdash_{DQL_{\mathcal{N}}} H(m) \to \langle\alpha^*\rangle H(0)$. Zusammen mit (ii) und den Regeln (G) und (MP) folgt daraus $\vdash_{DQL_{\mathcal{N}}} H(m) \to \langle\alpha^*\rangle G$ und schließlich mit (G) $\vdash_{DQL_{\mathcal{N}}} \forall m(H(m) \to \langle\alpha^*\rangle G)$. Durch Umformungen und Einbeziehung von (i) erhält man die Behauptung $\vdash_{DQL_{\mathcal{N}}} F \to \langle\alpha^*\rangle G$.
Bleibt noch zu zeigen, daß (i), (ii) und (iii) tatsächlich ableitbar sind. $F \to \exists m H(m)$ und $H(0) \to G$ sind in **N** gültige Formeln der Prädikatenlogik erster Stufe und mit (A1) gilt somit (i) und (ii). Es ist auch leicht zu sehen, daß (iii) gilt. Es wäre aber sehr mühsam, die Ableitung formal durchzuführen. Das liegt, wie schon erwähnt, daran, daß ein Hilbert-Kalkül nur wenige Ableitungsregeln besitzt.

4.5 Varianten der Dynamischen Logik

Wie schon erwähnt, gibt es nicht *die* Dynamische Logik. Verändert man die Menge der zulässigen Programme, so erhält man verschiedene Varianten der Dynamischen Logik. In diesem Abschnitt sollen einige Varianten der Dynamischen Logik erwähnt werden, ohne allerdings genauer darauf einzugehen. Eine ausführliche Betrachtung findet man zum Beispiel in [Hare84], wo auch auf die Ausdrucksstärke[11] der einzelnen Varianten eingegangen wird.

- **Zuweisung von zufälligen Werten:** Bei dieser Variante gibt es ein Programmkonstrukt, das Variablen zufällige Werte zuweist. Dadurch ist es möglich, Quantoren durch solche Konstrukte äquivalent zu ersetzen. Diese Variante ist ausdrucksstärker als DQL.

- **Rich-Test:** Hierbei sind Tests nicht mehr länger auf prädikatenlogische Formeln beschränkt, sondern es sind auch Tests von Programmen möglich. Damit kann ein Programm α anhalten, eine Frage stellen wie „kann ein Programm β mit der Eingabe x terminieren, wenn es jetzt gestartet würde? “

[11] Logik B ist ausdrucksstärker als Logik A, wenn es eine Formel F in B gibt, so daß für keine Formel G in A gilt: $F \leftrightarrow G$ ist gültig in B.

und dann ohne Seiteneffekte fortfahren, gdw. die Antwort „ja“ war. Erstaunlicherweise ist nicht bekannt, ob diese ziemlich mächtig erscheinende Variante tatsächlich ausdrucksstärker ist als DQL.

- **Rekursiv aufzählbare Dynamische Logik (r.a. DL):** Die zulässigen Programme der Dynamischen Logik, wie sie in den Abschnitten 4.1 und 4.2 definiert wurde, können als reguläre Ausdrücke über einem Alphabet, das aus den atomaren Programmen besteht, aufgefaßt werden. Eine viel größere Klasse von Programmen erhält man dadurch, daß man die zulässigen Programme als die rekursiv aufzählbare Zeichenketten über dem selben Alphabet definiert. Man braucht dann allerdings auch ein geeignetes Mittel, um diese Programme zu repräsentieren. Eine Möglichkeit ist die Beschreibung durch Turingmaschinen, die die zulässigen Programme erkennen. Solche Programme sind allerdings so komplex, daß bereits die Entscheidung, was ein Programm als nächstes tut, die volle Ausdrucksstärke einer Turingmaschine erfordert. Es kann gezeigt werden, daß diese Variante der Dynamischen Logik der Logik „constructive $L\omega_1\omega$“ sehr nahe kommt. Diese Logik entspricht ungefähr der Prädikatenlogik erster Stufe mit unendlichen Konjunktionen und Disjunktionen.

5 Programmiersprache

In diesem Abschnitt wird eine deterministische Programmiersprache, genannt **P**, definiert. Dazu wird zuerst eine weitere Variante der Dynamischen Quantorenlogik, nämlich die Deterministische Dynamische Quantorenlogik (DDQL) eingeführt. Die im Anschluß daran definierte Programmiersprache wird die in DDQL erlaubten Programme umfassen.

5.1 Deterministische Dynamische Quantorenlogik (DDQL)

Obwohl es durchaus Bereiche gibt, in denen man den Nichtdeterminismus zur Modellierung benötigt (zum Beispiel multitasking-fähige Betriebssysteme oder parallele Algorithmen), ist es doch so, daß in der Praxis die meisten Programme deterministisch sind. Im folgenden wird gezeigt, wie man durch ein paar einfache Modifikationen zu einer deterministischen Version der Dynamischen Quantorenlogik gelangt, bezeichnet als DDQL.
Der Indeterminismus der in den Kapiteln 4.1 und 4.2 definierten Dynamischen Logik resultiert einerseits aus den Programmkonstrukten $\cup$ und * und andererseits aus der Interpretation der atomaren Programme als binäre Relationen über Zuständen.
Dadurch, daß die atomaren Programme bei der Dynamischen Quantorenlogik, sprich die Zuweisungen, per Definition bereits deterministisch sind, ist diese Quelle des Indeterminismus verschwunden.
Den Indeterminismus der Programmkonstrukte $\cup$ und * kann man dadurch beseitigen, daß man diese Konstrukte nur noch in folgender Form erlaubt:

$$(P?;\alpha) \cup ((\neg P)?;\beta) \tag{1}$$

$$((P?;\alpha)^*;(\neg P)?). \tag{2}$$

Diese Restriktionen haben zur Folge, daß die Auswahl bei $\cup$ bzw. die Anzahl der Iterationen bei * durch die Auswertung der prädikatenlogischen Formel P gesteuert wird. Diese zwei Konstrukte entsprechen dann semantisch den aus vielen Programmiersprachen bekannten Kontrollstrukturen

$$\texttt{IF } P \texttt{ THEN } \alpha \texttt{ ELSE } \beta$$
$$\texttt{WHILE } P \texttt{ DO } \alpha.$$

Mit diesen beiden Einschränkungen erhält man als Menge der zulässigen Programme die bekannten *while-Programme.* Die Semantik bleibt dieselbe wie bei der Dynamischen Quantorenlogik.
Das folgende Lemma 19 zeigt, daß durch die Einschränkungen (1) und (2) der Syntax auch die zweite Quelle des Indeterminismus verschwunden ist. Dadurch sind

keine Relationen auf Zuständen zur Interpretation von Programmen mehr nötig, sondern es genügen Funktionen.

Lemma 19.
Für eine beliebige Struktur $\mathcal{M} = (U_{\mathcal{M}}, I_{\mathcal{M}})$*, einen Zustand* $v \in W$ *und ein deterministisches Programm* α *gilt:*
Es gibt höchstens einen Zustand $w \in W$ *mit* $(v, w) \in \rho(\alpha)$*.*

Aus diesem Lemma folgt sofort

Theorem 20.
Sei α *ein deterministisches Programm. Dann gilt*

$$\langle\alpha\rangle F \to [\alpha]F$$

Anschaulich gesprochen drückt dieses Theorem die Tatsache aus, daß ein deterministisches Programm, falls es terminiert, bei jeder Programmausführung immer im gleichen Endzustand terminiert.

5.2 Syntax

Im folgenden wird die Syntax der Programmiersprache **P** definiert. Dabei wird auf eine formale Definition der Sprache verzichtet. Der einzige Datentyp, der in **P** verwendet wird, ist **N**, die Menge der natürlichen Zahlen. Der Grund dafür ist, daß **N** mit den Operatoren „+" und „∗" eine arithmetische Struktur darstellt, die wohl bekannt ist.

Definition 21 (Reservierte Wörter).
Die Programmiersprache **P** *besitzt folgende reservierte Wörter, die nicht als Variablen oder verwendet werden dürfen:*
`IF, THEN, ELSE, WHILE.`

Definition 22 (Atomare Programme).
Ist x *eine Variablen und* t *ein Term. Dann ist folgendes Programmkonstrukt ein atomares Programme:*

$$x := t$$

Definition 23 (Programme).
„Bedingung" sei ein Boolscher Ausdruck. Dann werden Programme induktiv folgendermaßen definiert:

- *Atomare Programme sind Programme.*
- *Sind A,B Programme, dann ist* `IF` Bedingung A `ELSE` B *ein Programm. Der Teil „*`ELSE` B*" ist optional.*

- *Ist A ein Programm, dann ist* `WHILE` Bedingung *A ein Programm.*
- *Sind A,B Programme, dann ist A;B ein Programm.*

6 Prozeßlogik

6.1 Motivation und Entstehung

Wie bereits erwähnt wurde, ist die Dynamische Logik geeignet, Aussagen über das Eingabe-/Ausgabeverhalten von Programmen zu machen. Es ist aber nicht möglich, Aussagen über das Verhalten des Programms während der Ausführung zu formulieren. Logiken, die das ermöglichen, sind verschiedene Varianten der Temporallogik, zum Beispiel die von Pnuelli definierte Temporallogik für Programme [Pnue77, Pnue79]. In dieser Logik lassen sich Probleme wie *Verklemmungsfreiheit* oder *mutual exclusion* bei parallen Programmen ohne weiteres ausdrücken.
Allgemein läßt sich über Temporallogiken [Burg84] sagen, daß sie, ebenso wie die Dynamische Logik, Ideen der Modallogik aufgreifen. Die Temporallogik hat durchaus Ähnlichkeit mit der Dynamischen Logik. Sie betont das zeitliche Verhältnis von Zuständen und damit den Aussagen, die in diesen Zuständen wahr oder falsch sind. So sind Aussagen möglich wie „immer in der Zukunft (bzw. Vergangenheit) gilt...“ oder „manchmal in der Zukunft (bzw. Vergangenheit) gilt...“. Es gibt Varianten der Temporallogik, die nur diskrete Zeitpunkte zulassen, andere hingegen benutzen ein kontinuierliches Zeitspektrum. Desweiteren lassen sich lineare und verzweigende Temporallogiken unterscheiden. Bei ersteren gibt es nur einen Zeitstrahl und damit eine Zukunft, bei letzteren sind verschiedene „Zukunften“ möglich.
Das Problem bei allen Temporallogiken ist allerdings, daß das Programm als fest und zu der Struktur gehörig angesehen werden muß, in der die Logik interpretiert werden soll. Das Programm wird dadurch von der Syntax der Logik getrennt.[12] Dadurch verliert man den induktiven Aufbau von Formeln und Programmen wie in der Dynamischen Logik. Deshalb wurde die Prozeßlogik mit dem Ziel entwickelt, die Vorteile beider Logiken, also die größere Ausdrucksstärke der Temporallogik und den induktiven Formelaufbau der Dynamischen Logik, zu vereinen.
Dieses Kapitel steht im Mittelpunkt der ganzen Arbeit. Zuerst wird eine Prozeßaussagenlogik vorgestellt, die eine Erweiterung der Dynamischen Aussagenlogik darstellt [Prat79]. Im Anschluß daran soll eine Prozeßlogik erster Stufe für die in Kapitel 5 festgelegte Programmiersprache **P** definiert werden. Das Ziel ist schließlich eine Axiomatisierung der Prozeßlogik erster Stufe in einem Sequenzenkalkül.

[12] In [KoTi90] werden solche Logiken *endogen* genannt. Im Gegensatz dazu stehen die *exogenen* Logiken, zum Beispiel die Dynamische Logik, bei denen die Programme explizit Teil der Sprache sind.

6.2 Prozeßaussagenlogik

6.2.1 Syntax

Die Definition der Syntax stimmt weitestgehend mit der Syntax der Dynamische Aussagenlogik überein. Zusätzlich zu $\langle\cdot\rangle$ und $[\cdot]$ stehen folgende Modalitäten zur Verfügung:

$\subset \cdot \supset$	„durchgehend gilt ...“
$\ll \cdot \gg$	„mindestens einmal gilt ...“
$\| \cdot \|$	„ab einem Punkt gilt durchgehend ...“

Definition 24 (Syntax der Prozeßaussagenlogik).
Eine Signatur $\Sigma = \langle \Pi_0, F_0 \rangle$ *besteht aus einer nichtleeren Menge* Π_0 *atomarer Programme und einer Menge* F_0 *aussagenlogischer Variablen. Die Menge der Formeln der Prozeßaussagenlogik* Fml_{pal} *und die Menge* Π *der erlaubten Programme wird simultan induktiv definiert durch:*

- *true, false* $\in Fml_{pal}$,
- $F_0 \subseteq Fml_{pal}$,
- *sind* F_1, $F_2 \in Fml_{pal}$, *dann auch* $\neg F_1$, $(F_1 \vee F_2)$,
- $\Pi_0 \subseteq \Pi$,
- *sind* $\alpha, \beta \in \Pi$, *dann auch* $(\alpha;\beta)$, $(\alpha \cup \beta)$ *und* (α^*),
- *sind* $F \in Fml_{pal}$ *und* $\alpha \in \Pi$, *dann gilt* $\langle\alpha\rangle F \in Fml_{pal}$,
- *sind* $F \in Fml_{pal}$ *und* $\alpha \in \Pi$, *dann gilt* $\subset\alpha\supset F \in Fml_{pal}$,
- *sind* $F \in Fml_{pal}$ *und* $\alpha \in \Pi$, *dann gilt* $\ll\alpha\gg F \in Fml_{pal}$,
- *sind* $F \in Fml_{pal}$ *und* $\alpha \in \Pi$, *dann gilt* $\|\alpha\| F \in Fml_{pal}$,
- *sind* $F \in Fml_{pal}$ *und* F *programmfrei, dann gilt* $(F?) \in \Pi$.

Es werden dieselben Abkürzungen wie bei der Dynamischen Aussagenlogik benutzt, insbesondere $[\alpha]F$ für $\neg\langle\alpha\rangle\neg F$.

6.2.2 Semantik

Um Formeln der Prozeßaussagenlogik interpretieren zu können, ist, wie bei der Dynamischen Aussagenlogik, eine Struktur erforderlich. Der Unterschied zwischen einer Struktur der Prozeßaussagenlogik und einer Struktur der Dynamischen Aussagenlogik ist, daß die atomaren Programme der Prozeßaussagenlogik nicht als binäre Relation interpretiert werden, sondern als *Zustandsfolgen.*

Definition 25 (Zustandsfolge ZF).
Sei W eine nichtleere Menge von Zuständen. Eine Folge $s \subseteq W$ von Zuständen heißt Zustandsfolge (ZF). Ist $s = (s_0, s_1, \ldots, s_m)$ endlich, dann ist m die Länge der Zustandsfolge und s_m ist der Endzustand der Zustandsfolge.

Definition 26 (Zustandsfolge ZF bei „deadlock", „abort" und „fail").
Sei $W = W' \cup \{\Lambda\}$ eine nichtleere Menge von Zuständen. Λ ist ein reservierter Zustand, der nur am Ende einer Zustandsfolge vorkommen kann und dann ein Fehlschlagen („deadlock", „abort" und „fail") der Programmabarbeitung kennzeichnet.[13] *Eine Zustandsfolge ist genau dann eine terminierende Zustandsfolge, wenn ein Endzustand existiert und ungleich Λ ist.*

In der Folge bezeichne W immer eine Menge an Zuständen, die Λ enthält, und W' sei $W \setminus \{\Lambda\}$.

Definition 27 (Folgen der Länge Null).
Sei W eine Menge von Zuständen. Dann wird die Menge der Zustandsfolgen I der Länge Null definiert durch $I = \{(w) \mid w \in W\}$.

Definition 28 (PAL-Struktur).
Eine Struktur $\mathcal{B} = (W, \tau, \phi)$ für die Prozeßaussagenlogik besteht aus

- *einer nichtleeren Menge von Zuständen W (Λ sei enthalten),*
- *einer Interpretation der aussagenlogischen Variablen, d.h., einer Funktion $\tau : F_0 \to W$,*
- *einer Interpretation ϕ*[14] *der atomaren Programme, d.h., einer Funktion $\phi : \Pi_0 \to W!$. Dabei bezeichnet $W!$ die Menge aller Zustandsfolgen über W.*

Bevor die Semantik der Prozeßaussagenlogik definiert werden kann, sind noch einige Hilfsdefinitionen nötig.

[13] Pratt nennt diesen Zustand „limbo", zu deutsch „Vorhölle".
[14] Bei einer DAL-Struktur wurde der Buchstabe ρ zur Interpretation der Programme als binäre Relationen verwendet.

Definition 29 (Präfix einer Zustandsfolge).
Sei $s = (s_0, s_1, \ldots, s_k, \ldots)$ eine Zustandsfolge mit $s_k \neq \Lambda$. s_k kann sowohl ein Endzustand als auch ein Nichtendzustand sein. Dann ist die Zustandsfolge $s = (s_0, s_1, \ldots, s_k, \Lambda)$ ein Präfix von s.

Definition 30 (Verkettung von Zustandsfolgen).
Seien $s = (s_0, s_s, \ldots, s_m)$ und $t = (t_0, t_1, \ldots, t_n)$ zwei Zustandsfolgen. Ist s endlich, $s_m \neq \Lambda$ und $s_m = t_0$ dann ist die Verkettung $s;t$ von s und t definiert durch die Zustandsfolge

$$(s_0, s_1, \ldots, s_m, t_1, t_2, \ldots, t_n).$$

Ist s unendlich, dann ist $s;t$ definiert als s.
Andernfalls (s_m ist endlich und $s_m \neq t_0$) ist $s;t$ nicht definiert.

Definition 31 (Verkettungsprodukt).
Seien S und T zwei Mengen von Zustandsfolgen. Dann ist das Verkettungsprodukt $S;T$ von S und T definiert durch

$$\{s;t \mid s \in S, s \in T \text{ und } s;t \text{ ist definiert } \}.$$

Definition 32 (Präfixfreie Zustandsfolgenmenge).
Eine Zustandsfolgenmenge S ist genau dann präfixfrei, wenn keine Zustandsfolge $s \in S$ Präfix einer anderen Zustandsfolge $t \in S$ ist.

Definition 33 (Präfixfreier Abschluß).
Der präfixfreie Abschluß $CL_{pf}(S)$ einer Zustandsfolgenmenge S ist die größte präfixfreie Zustandsfolgenmenge $S' \subseteq S$. Man erhält $CL_{pf}(S)$, indem man aus S nacheinander alle Zustandsfolgen entfernt, die Präfix einer anderen Zustandsfolge aus S sind.

Definition 34 (Semantik für die Prozeßaussagelogik).
Sei $\mathcal{B} = (W, \tau, \phi)$ eine Struktur für die Prozeßaussagenlogik und $W' = W \setminus \{\Lambda\}$, dann wird die Funktion τ fortgesetzt auf die Menge aller Formel F und ϕ wird fortgesetzt auf die Menge aller Programme Π durch folgende simultane induktive Definition.

- $\tau(\mathit{true}) = W'$ *und* $\tau(\mathit{false}) = \emptyset$,
- $\tau(\neg F) = W' \setminus \tau(F)$,
- $\tau(F_1 \vee F_2) = \tau(F_1) \cup \tau(F_2)$,
- $\tau(\langle\alpha\rangle F) = \{s_0 \in W' \mid$ *es gibt ein* $s \in \phi(\alpha)$ *mit* $s = (s_0, s_1, \ldots, s_m)$, $s_m \neq \Lambda$ *und* $s_m \in \tau(F)\}$,

- $\tau(\subset\alpha\supset F) = \{s_0 \in W' \mid$ *für alle* $s \in \phi(\alpha)$ *mit* $s = (s_0, s_1, \ldots, s_m)$ *gilt: für alle* $s_i \neq \Lambda$ *gilt:* $s_i \in \tau(F)\}$,
- $\tau(\ll\alpha\gg F) = \{s_0 \in W' \mid$ *es gibt ein* $s \in \phi(\alpha)$ *mit* $s = (s_0, s_1, \ldots, s_m)$ *und es gibt ein* $s_i \neq \Lambda$ *mit* $s_i \in \tau(F)\}$,
- $\tau(\|\alpha\|F) = \{s_0 \in W' \mid$ *für alle* $s \in \phi(\alpha)$ *mit* $s = (s_0, s_1, \ldots, s_m)$ *gilt: für alle* $s_i \neq \Lambda, s_j \neq \Lambda$ *gilt: wenn* $i < j$ *und* $s_i \in \tau(F)$ *dann ist auch* $s_j \in \tau(F)\}$,
- $\rho(\alpha;\beta) = CL_{pf}(\rho(\alpha);\rho(\beta))$,
- $\rho(\alpha \cup \beta) = CL_{pf}(\rho(\alpha) \cup \rho(\beta))$,
- $\rho(\alpha^*) = \rho(\alpha)^* = CL_{pf}(I \cup \rho(\alpha) \cup (\rho(\alpha);\rho(\alpha)) \cup \ldots \cup (\phi(\alpha);\phi(\alpha);\ldots))$,
- $\rho(F?) = CL_{pf}(\{(w) \mid w \in \tau(F)\} \cup \{(w,\Lambda) \mid w \notin \tau(F)\} \cup \{(\Lambda)\})$.

Erfüllbarkeit, Gültigkeit und Allgemeingültigkeit einer Formel sind definiert wie in Abschnitt 4.1.
Die Definition der Semantik zeigt, daß die Modalität $\subset\cdot\supset$ durch $\ll\cdot\gg$ ausdrückbar ist und umgekehrt. So sind die beiden Formeln $\subset\alpha\supset\varphi$ und $\neg\ll\alpha\gg\neg\varphi$ äquivalent.

6.2.3 Anmerkungen zur Semantik

Im Vergleich zur Semantik der Dynamischen Aussagenlogik fallen bei der Interpretation der Programme zwei Dinge auf: erstens die Verwendung des präfixfreien Abschlusses CL_{pf} und zweitens eine andere Interpretation von Tests.
CL_{pf} wird verwendet, um zu verhindern, daß ein Programm einen Test ausführt, der fehlschlägt, wenn es alternativ auch eine andere Fortsetzung des Programms gibt. Dazu ist es erforderlich, bei der Interpretation von Tests auch die Zustände aufzuzählen, in denen die zu testende Formel nicht erfüllt ist. Diese werden dann mit dem Sonderzustand Λ gekennzeichnet. Auch bei den in der Einleitung erwähnten Computation Trees [Hare79], Kapitel 5, wird ein ähnlicher Ansatz verfolgt.
An folgendem Beispiel soll demonstriert werden, was der präfixfreie Abschluß bewirkt und daß die Semantik der Konstrukte $(P?;\alpha)\cup((\neg P)?;\beta)$ und $(P?;\alpha)^*;(\neg P)?$ tatsächlich mit der intendierten Semantik von „`IF` P `THEN` α `ELSE` β“ und „`WHILE` P `DO` α“ übereinstimmt.

Beispiel: Gegeben sei das Programm

$$(P?;\alpha) \cup ((\neg P)?;\beta)$$

und die PAL-Struktur $\mathcal{B} = (W, \tau, \phi)$ mit

$$W = \{1,2,3,4,\Lambda\}$$
$$\phi(\alpha) = \{(1,2)\}$$
$$\phi(\beta) = \{(1,4)\}$$
$$\tau(P) = \{1\}$$

$\tau(P) = \{1\}$ bedeutet, daß im Zustand 1 die Formel P gilt. Deshalb erwartet man, daß das Programm α ausgeführt wird und nicht das Programm β, obwohl dieses auch im Zustand 1 startet.
Mit der definierten Semantik ergibt sich dann

$$\phi((P?;\alpha) \cup ((\neg P)?;\beta)) = CL_{pf}(\phi(P?;\alpha) \cup \phi((\neg P)?;\beta))) =$$
$$CL_{pf}(\phi(P?);\phi(\alpha) \cup \phi((\neg P)?);\phi(\beta))) =$$
$$CL_{pf}((\{(1),(2,\Lambda),(3,\Lambda),(4,\Lambda),(\Lambda)\};\{(1,2)\}) \cup (\{(1,\Lambda),(2),(3),(4),(\Lambda)\};\{(1,4)\})) =$$
$$\{(1,2)\}$$

Man sieht, daß tatsächlich nur das Programm α ausgeführt werden kann. Nun ändert man die Funktion τ ab zu $\tau(P) = \{2\}$, d.h. die Formel P gilt im Zustand 1 nicht mehr und man erwartet nun, daß nur das Programm β ausgeführt werden kann.

$$\phi((P?;\alpha) \cup ((\neg P)?;\beta)) = CL_{pf}(\phi(P?;\alpha) \cup \phi((\neg P)?;\beta))) =$$
$$CL_{pf}(\phi(P?);\phi(\alpha) \cup \phi((\neg P)?);\phi(\beta))) =$$
$$CL_{pf}((\{(1,\Lambda),(2),(3,\Lambda),(4,\Lambda),(\Lambda)\};\{(1,2)\}) \cup (\{(1),(2,\Lambda),(3),(4),(\Lambda)\};\{(1,4)\})) =$$
$$\{(1,4)\}$$

Man sieht, daß die Ausführung des Programmes $(P?;\alpha) \cup ((\neg P)?;\beta)$ tatsächlich deterministisch ist. Mit der Interpretation der Formel P kann man steuern, ob das Programm α oder das Programm β ausgeführt werden soll.
Der präfixfreie Abschluß hatte bei diesem Beispiel noch keine Auswirkung. Im nächsten Beispiel jedoch wird man sehen, daß mit Hilfe von CL_{pf} die „unerwünschten“ Möglichkeiten zur Programmausführung beseitigt werden.

Beispiel: Gegeben sei das Programm

$$\beta = (P?;\alpha)^*;(\neg P)?$$

und die PAL-Struktur $\mathcal{B} = (W,\tau,\phi)$ mit

$$W = \{1,2,3,4,\Lambda\}$$
$$\tau(P) = \{1,2,3\}$$
$$\phi(\alpha) = \{(1,2);(2,3);(3,4)\}$$

Die Zustandsfolgen, die man von der Programmausführung erwartet, wenn man die Semantik von „`WHILE` P `DO` α" im Kopf hat, sind diejenigen Folgen, die in $s = \{(1,2,3,4); (2,3,4); (3,4); (4)\}$ enthalten sind. In welchen von diesen β wahr ist, hängt davon ab, in welchem Zustand man startet.

Abbildung 5 zeigt alle möglichen Zustandsfolgen, die man ohne den präfixfreien Abschluß erhält. Die Knoten repräsentieren die Zustände, die Kanten sind mit jeweils dem (Teil-)Programm beschriftet, mit dem man in den entsprechenden Zustand gelangt. Man sieht, daß alle „unerwünschten" Zustandsfolgen Präfixe anderer Zustandsfolgen sind und deshalb durch den Operator CL_{pf} beseitigt werden. So ist beispielsweise $(1,2,3,\Lambda)$ Präfix von $(1,2,3)$.

6.2.4 Beispiele

Nach den Beispielen im vorigen Abschnitt sollen hier noch einige äquivalenzen prozeßaussagenlogischer Formeln angegeben werden (Abbildung 6).

6.3 Prozeßlogik erster Stufe

Die im letzten Abschnitt definierte Prozeßaussagenlogik ist eher von theoretischem Interesse. Will man die Prozeßlogik in der Praxis einsetzen, das heißt zum Beispiel zur Verifikation von Programmen, dann muß man zu einer Prozeßlogik erster Stufe übergehen, die Quantoren und reale Programmkonstrukte (Zuweisungen) enthält. Dieser Übergang ist analog zu dem Übergang von der Dynamischen Aussagenlogik zur Dynamischen Quantorenlogik: es werden Quantoren zugelassen und die atomaren Programme der Aussagenlogik verschwinden. An deren Stelle tritt bei der Prozeßlogik erster Stufe die Zuweisung als einziges elementares Programmkonstrukt. Von großer Bedeutung dabei ist, daß die Semantik der Zuweisung fest definiert ist. Im Gegensatz dazu muß man die Semantik der atomaren Programme der Prozeßaussagenlogik selbst definieren, d.h., die Semantik der atomaren Programme ist Teil der Struktur.

6.3.1 Syntax

Definition 35 (Formeln und Programme).
Eine Signatur $\Sigma = \langle P, F\rangle$ *für die Prozeßlogik erster Stufe besteht aus einer Menge* P *der Prädikatszeichen und einer Menge* F *der Funktionszeichen. Die Menge der Individuenvariablen sei mit* Var *bezeichnet. Die Menge der Formeln* Fml_{pl} *der Prozeßlogik erster Stufe und der erlaubten Programme* Π *wird simultan induktiv definiert durch*

- *true, false* $\in Fml_{pl}$,

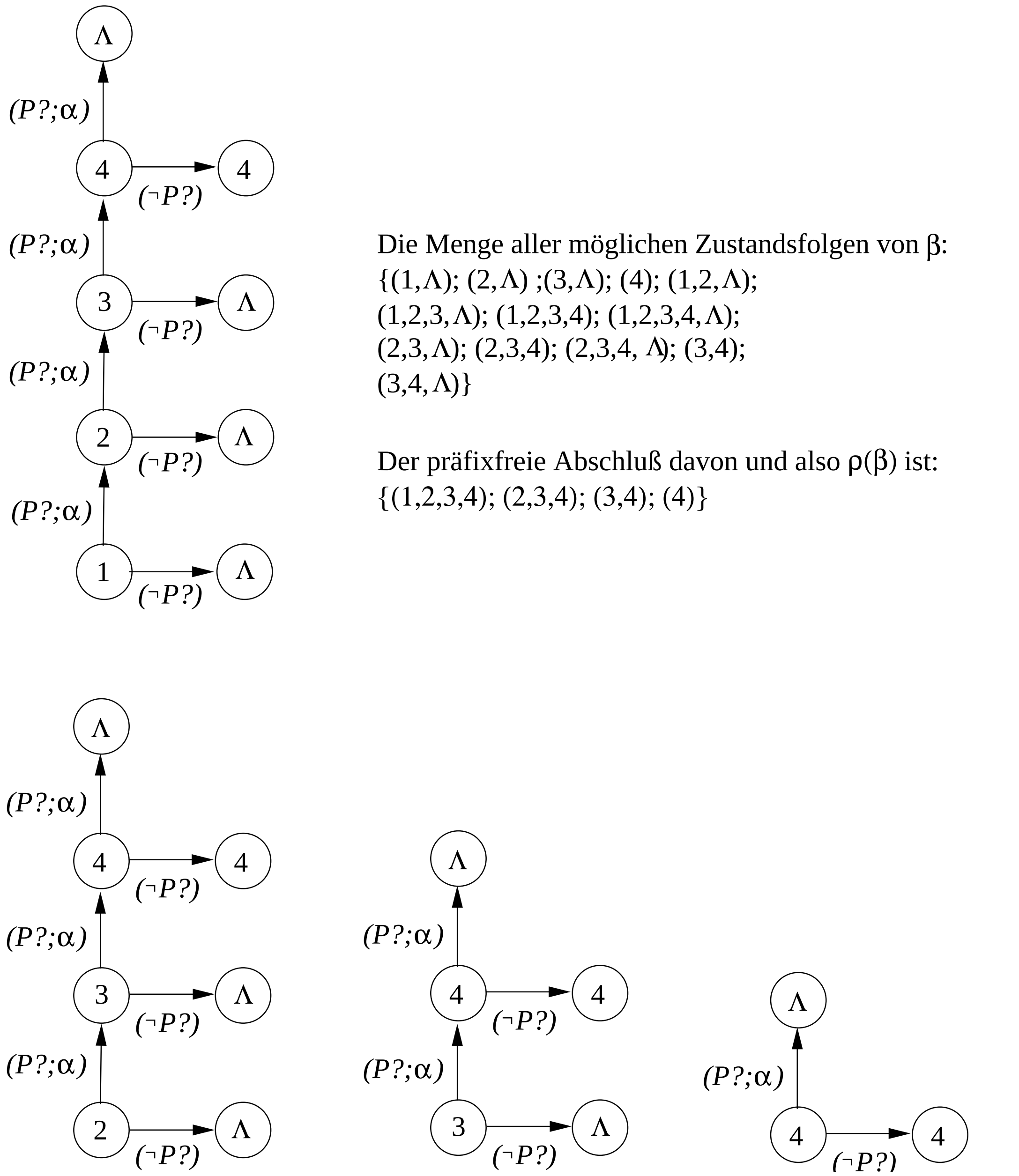

Abbildung 5: Mögliche Zustandsfolgen des Programms $\beta = (P?;\alpha)^*;(\neg P)?$

- *jede atomare Formel ist in* Fml_{pl},
- *sind* F_1, $F_2 \in Fml_{pl}$ *und ist* x *eine Variable, dann sind auch* $\neg F_1$, $(F_1 \vee F_2)$, $\exists x F_1 \in Fml_{pl}$,

$[\alpha \cup \beta]F$	$\equiv$	$[\alpha]F \wedge [\beta]F$	$\subset\alpha \cup \beta\supset F$	$\equiv$	$\subset\alpha\supset F \wedge \subset\beta\supset F$
$[\alpha;\beta]F$	$\equiv$	$[\alpha][\beta]F$	$\subset\alpha;\beta\supset F$	$\equiv$	$\subset\alpha\supset F \wedge [\alpha]\subset\beta\supset F$
$[\alpha^*]F$	$\equiv$	$F \wedge [\alpha][\alpha^*]F$	$\subset\alpha^*\supset F$	$\equiv$	$[\alpha^*]\subset\alpha\supset F$
$[G?]F$	$\equiv$	$G \rightarrow F$	$\subset G?\supset F$	$\equiv$	F
$\|\alpha \cup \beta\|F$	$\equiv$	$\|\alpha\|F \wedge \|\beta\|F$	$\ll\alpha \cup \beta\gg F$	$\equiv$	$\ll\alpha\gg F \wedge \ll\beta\gg F$
$\|\alpha;\beta\|F$	$\equiv$	$\|\alpha\|F \wedge [\alpha]\|\beta\|F$	$\ll\alpha;\beta\gg F$	$\equiv$	$\ll\alpha\gg F \vee \langle\alpha\rangle\ll\beta\gg F$
$\|\alpha^*\|F$	$\equiv$	$[\alpha^*]\|\alpha\|F$	$\ll\alpha^*\gg F$	$\equiv$	$\langle\alpha^*\rangle F$
$\|G?\|F$	$\equiv$	$true$	$\ll G?\gg F$	$\equiv$	F

Abbildung 6: Äquivalenzen prozeßaussagenlogischer Formeln

- *für jede Variable x und jeden Term t ist $(x := t) \in \Pi$,*
- *für $F \in Fml_{pl}$ gilt $(F?) \in \Pi$, falls F programmfrei,*
- *sind $\alpha, \beta \in \Pi$, dann auch $(\alpha;\beta)$, $(\alpha \cup \beta)$ und (α^*),*
- *für $F \in Fml_{pl}$ und $\alpha \in \Pi$ gilt $\langle\alpha\rangle F \in Fml_{pl}$,*
- *für $F \in Fml_{pl}$ und $\alpha \in \Pi$ gilt $\|\alpha\|F \in Fml_{pl}$,*
- *für $F \in Fml_{pl}$ und $\alpha \in \Pi$ gilt $\subset\alpha\supset F \in Fml_{pl}$,*
- *für $F \in Fml_{pl}$ und $\alpha \in \Pi$ gilt $\ll\alpha\gg F \in Fml_{pl}$.*

Wie bei der Prozeßaussagenlogik werden überflüssige Klammern oft weggelassen. Ferner werden wieder die üblichen Abkürzungen $[\alpha], \wedge, \rightarrow, \leftrightarrow$ verwendet.

6.4 Semantik

Die Verkettung „;“ von Zustandsfolgen ist wie im aussagenlogischen Fall definiert. Dasselbe gilt für den präfixfreien Abschluß CL_{pf}.

Definition 36 (Semantik für die Prozeßlogik erster Stufe).
Sei $\mathcal{M} = (U_{\mathcal{M}}, I_{\mathcal{M}})$ eine Struktur für die Prädiaktenlogik erster Stufe und $W' = \{w \mid w : Var \rightarrow U_{\mathcal{M}}\}$ die Menge der Zustände, dann wird die Semantik der Prozeßlogik erster Stufe simultan induktiv definiert durch (definiert werden ϕ und τ):

- *$w \in \tau(R(t_1, \ldots, t_n))$ gdw. die atomare Formel $R(t_1, \ldots, t_n)$ ist wahr in der Struktur $\mathcal{M}$ unter der Belegung der Variablen w,*

- $\phi(x := t) = \{(w_1, w_2) \mid w_1 \neq \Lambda, w_2 \neq \Lambda, w_2(y) = w_1(y)$ *für* $y \neq x$ *und* $w_2(x) = I(\mathcal{M}, w_1)(t)\}$, *wobei* $I(\mathcal{M}, w_1)(t)$ *die Interpretation des Terms* t *unter der Belegung* w_1 *ist,*
- $\tau(true) = W'$ *und* $\tau(false) = \emptyset$,
- $\tau(\neg F) = W' \setminus \tau(F)$,
- $\tau(F_1 \vee F_2) = \tau(F_1) \cup \tau(F_2)$,
- $\tau(\exists x F) = \{w \mid$ *es existiert ein* $d \in U_{\mathcal{M}}$ *mit* $w_x^d \in \tau(F)\}$
- $\tau(\langle\alpha\rangle F) = \{s_0 \in W \mid$ *es gibt ein* $s \in \phi(\alpha)$ *mit* $s = (s_0, s_1, \ldots, s_m)$, $s_m \neq \Lambda$ *und* $s_{m_S} \in \tau(F)\}$,
- $\tau(\subset\alpha\supset F) = \{s_0 \in W \mid$ *für alle* $s \in \phi(\alpha)$ *gilt: für alle* $s_i \in s, s_i \neq \Lambda$ *gilt:* $s_i \in \tau(F)\}$,
- $\tau(\ll\alpha\gg F) = \{s_0 \in W \mid$ *es gibt ein* $s \in \phi(\alpha)$ *und es gibt ein* $s_i \in s, s_i \neq \Lambda$ *mit* $s_i \in \tau(F)\}$,
- $\tau(\|\alpha\| F) = \{s_0 \in W \mid$ *für alle* $s \in \phi(\alpha)$ *gilt: für alle* $s_i \in s, s_j \in s$ *mit:* $s_i \neq \Lambda, s_j \neq \Lambda$ *und* $i < j$ *gilt: wenn* $s_i \in \tau(F)$, *dann auch* $s_j \in \tau(F)\}$,
- $\phi(\alpha; \beta) = CL_{pf}(\phi(\alpha); \phi(\beta))$,
- $\phi(\alpha \cup \beta) = CL_{pf}(\phi(\alpha) \cup \phi(\beta))$,
- $\phi(\alpha^*) = \phi(\alpha)^* = CL_{pf}(I \cup \phi(\alpha) \cup (\phi(\alpha); \phi(\alpha)) \cup \ldots \cup (\phi(\alpha); \phi(\alpha); \ldots))$,
- $\phi(F?) = CL_{pf}(\{(w) \mid w \in \tau(F)\} \cup \{(w, \Lambda) \mid w \notin \tau(F)\} \cup \{(\Lambda)\})$.

Erfüllbarkeit, Gültigkeit und Allgemeingültigkeit einer Formel sind definiert wie in Abschnitt 4.2.
Der Unterschied zur Semantik der Prozeßaussagenlogik besteht lediglich darin, daß W festgelegt wird auf die Variablenbelegungen und daß die Semantik für die Zuweisung und die Quantoren definiert wird, die im aussagenlogischen Fall ja nicht existieren.

7 Ableitungsregeln für einen Sequenzenkalkül

In diesem Abschnitt werden Ableitungsregeln für einen Sequenzenkalkül für die deterministische[15] Programmiersprache **P** angegeben. Um aber tatsächlich Ableitungen vornehmen zu können, braucht man noch einige für einen Sequenzenkalkül typische Regeln, die unabhängig von den speziellen Regeln für **P** sind.
Taucht in einer Regel die Variable x' bzw. der Vektor von Variablen $\vec{x}'$ auf, dann sollen diese stets neue, noch nicht verwendete Variablen sein.

7.1 Allgemeine Regeln

- Einführung von Tautologien als Axiome (R1)

  $$\frac{}{\vdash \ \varphi} \ \varphi \text{ sei eine aussagenlogische Tautologie}$$

- Einführung von gültigen prädikatenlogischen Formeln als Axiome (R2)

  $$\frac{}{\vdash \ \varphi} \ \varphi \text{ sei in arithmetischen Strukturen gültig.}$$

  Bemerkung: Diese Regel ist für die relative Vollständigkeit erforderlich. Man sollte aber beachten, daß die Gültigkeit von φ nicht berechenbar ist!

- $\forall$ rechts (R3)

  $$\frac{\Gamma \ \vdash \ \varphi_{\vec{x}}^{\vec{x}'}, \ \Delta}{\Gamma \ \vdash \ \forall \vec{x} \varphi, \ \Delta}$$

  Diese Regel ist auch anwendbar, wenn man Prämisse und Konklusion vertauscht. Die Regel ($\forall$ links) wird nicht explizit angegeben, da sie mit den Negationsregeln (R5), (R6) und ($\exists$ links) (R4) simuliert werden kann.

- $\exists$ links (R4)

  $$\frac{\Gamma, \ \varphi_{\vec{x}}^{\vec{x}'} \ \vdash \ \Delta}{\exists \vec{x} \varphi, \ \Gamma \ \vdash \ \Delta}$$

  Die Regel „$\exists$ rechts“ wird ebenfalls nicht angegeben, da sie analog zur Regel ($\forall$ links) simuliert werden kann.

[15] Es wäre wahrscheinlich auch möglich, einen Kalkül für nichtdeterministische Programme anzugeben, da in dieser Arbeit eine andere Semantik definiert wurde als in [Prat79], wo ein Kalkül nicht aufgestellt werden konnte. Allerdings ist für praktische Anwendungen eine Einschränkung auf deterministische Programme meist gerechtfertigt.

- Negation links (R5)
$$\frac{\Gamma,\ \varphi\ \vdash\ \Delta}{\Gamma\ \vdash\ \neg\varphi,\ \Delta}$$

- Negation rechts (R6)
$$\frac{\Gamma\ \vdash\ \Delta,\ \varphi}{\Gamma,\ \neg\varphi\ \vdash\ \Delta}$$

- Schnitt (R7)
$$\frac{\Gamma,\ \varphi\ \vdash\ \Delta \qquad \Gamma\ \vdash\ \varphi,\ \Delta}{\Gamma\ \vdash\ \Delta}\ \varphi\ \textit{beliebig}$$

- Abschwächung links (R8)
$$\frac{\Gamma\ \vdash\ \Delta}{\Gamma,\ \varphi\ \vdash\ \Delta}$$

- Abschwächung rechts (R9)
$$\frac{\Gamma\ \vdash\ \Delta}{\Gamma\ \vdash\ \Delta,\ \varphi}$$

- $\wedge$ links (R10)
$$\frac{\Gamma,\ \varphi,\ \psi\ \vdash\ \Delta}{\Gamma,\ \varphi\wedge\psi\ \vdash\ \Delta}$$

- $\wedge$ rechts (R11)
$$\frac{\Gamma\ \vdash\ \varphi,\ \Delta \qquad \Gamma\ \vdash\ \psi,\ \Delta}{\Gamma\ \vdash\ \varphi\wedge\psi}$$

- $\vee$ rechts (R12)
$$\frac{\Gamma\ \vdash\ \varphi,\ \psi,\ \Delta}{\Gamma\ \vdash\ \varphi\vee\psi,\ \Delta}$$

- $\vee$ links (R13)
$$\frac{\Gamma,\ \varphi\ \vdash\ \Delta \qquad \Gamma,\ \psi\ \vdash\ \Delta}{\Gamma,\ \varphi\vee\psi\ \vdash\ \Delta}$$

- Generalisierung' (R14)

$$\frac{\varphi \;\vdash\; \delta}{\exists x\varphi \;\vdash\; \exists x\delta}$$

- Induktion (R15)

$$\frac{\Gamma \;\vdash\; \varphi(0),\;\; \Delta \qquad\qquad \Gamma,\;\; \varphi(n) \;\vdash\; \varphi(n+1),\;\; \Delta}{\Gamma \;\vdash\; \forall n\varphi(n),\;\; \Delta}\; n \textit{ kommt in } \Gamma, \Delta \textit{ nicht vor}$$

7.2 Regeln für die Programmiersprache P

Nun folgen die Regeln für die Programmiersprache **P** . Die Regeln werden getrennt nach den vier Modalitäten angegeben.
Für die Zuweisungsregeln werden Variablenumbenennungen auf Formeln der Prozeßlogik angewandt. Eine Variablenumbenennung für die Prozeßlogik wird durch Fortsetzung der gewöhnlichen Variablenumbennenung, wie sie aus der Prädikatenlogik bekannt ist, definiert.

Definition 37 (Variablenumbenennung).
Sei $F, F_1, F_2 \in Fml_{pl}$*,* $\alpha, \beta \in \Pi$ *und* σ *eine Substitution mit* $\sigma(x) \in Var$ *für alle* $x \in Var$ *und* σ *injektiv, d.h.* σ *sei eine Variablenumbenennung. Dann wird* σ *induktiv auf Formeln der Prozeßlogik erster Stufe und auf Programme fortgesetzt durch*

$$\sigma : Fml_{pl} \to Fml_{pl}$$
$$\sigma : \Pi \to \Pi$$

mit

- $\sigma(true) = true, \sigma(false) = false$
- F *atomar:* $\sigma(F)$ *definiert wie in der Prädikatenlogik*
- $\sigma(F_1 \vee F_2) = \sigma(F_1) \vee \sigma(F_2), \sigma(\neg F) = \neg\sigma(F)$
- $\sigma(\exists x F) = \exists x \sigma_x(F)$ *mit* $\sigma_x(x) = x$ *und* $\sigma_x(y) = \sigma(y)$ *für* $y \neq x$
- $\sigma(x := t) = (\sigma(x) := \sigma(t))$
- $\sigma(F?) = (\sigma(F))?$
- $\sigma(\alpha; \beta) = (\sigma(\alpha); \sigma(\beta)), \sigma(\alpha \cup \beta) = (\sigma(\alpha) \cup \sigma(\beta)), \sigma(\alpha^*) = (\sigma(\alpha))^*$

- $\sigma(\langle\alpha\rangle F) = \sigma(\langle\alpha\rangle)\sigma(F)$
- $\sigma(\|\alpha\| F) = \sigma(\|\alpha\|)\sigma(F)$
- $\sigma(\subset\alpha\supset F) = \sigma(\subset\alpha\supset)\sigma(F)$
- $\sigma(\ll\alpha\gg F) = \sigma(\ll\alpha\gg)\sigma(F)$.

In den Regeln wird für die Substitution die Schreibweise $F_x^{x'}$ verwendet, was definiert ist als $\sigma(F)$ mit $\sigma = \{x \leftarrow x'\}$.

Definition 38 (Kollisionsfreie Variablenumbenennung).
Eine Variablenumbenennung σ auf Formeln der Prozeßlogik erster Stufe ist kollisionsfrei, wenn sie kollisionsfrei im Sinne der Prädikatenlogik ist.

7.2.1 Regeln für $\langle\cdot\rangle$ und $[\cdot]$

Die folgenden Regeln entsprechen den in KIV [Reif92] verwendeten Regeln für die Dynamische Logik.
$\langle\cdot\rangle$ drückt aus, daß es mindestens eine mögliche Programmausführung gibt, so daß danach eine gewisse Formel gilt. Im Gegensatz dazu sichert $[\cdot]$ die Gültigkeit einer Formel am Ende jeder möglichen terminierenden Programmausführung zu. Im deterministischen Fall ist der einzige Unterschied zwischen diesen beiden Modalitäten, daß $\langle\cdot\rangle$ die Terminierung garantiert und $[\cdot]$ nicht.

- Zuweisung rechts (R16)
  $$\frac{\Gamma_x^{x'},\ x = t_x^{x'} \ \vdash\ \varphi,\ \Delta_x^{x'}}{\Gamma\ \vdash\ \langle x := t\rangle\varphi,\ \Delta}$$
  Diese Regel ist analog auch auf $[\cdot]$ anwendbar.

- Zuweisung links (R17)
  $$\frac{\varphi,\ x = t_x^{x'},\ \Gamma_x^{x'}\ \vdash\ \Delta_x^{x'}}{\langle x := t\rangle\varphi,\ \Gamma\ \vdash\ \Delta}$$
  Diese Regel ist analog auch auf $[\cdot]$ anwendbar.

- ; rechts (R18)
  $$\frac{\Gamma\ \vdash\ \langle\alpha\rangle\langle\beta\rangle\varphi,\ \Delta}{\Gamma\ \vdash\ \langle\alpha;\beta\rangle\varphi,\ \Delta}$$
  Diese Regel ist analog auch auf $[\cdot]$ anwendbar.

- ; links (R19)

$$\frac{\Gamma,\ \langle\alpha\rangle\langle\beta\rangle\varphi\ \vdash\ \Delta}{\Gamma,\ \langle\alpha;\beta\rangle\varphi\ \vdash\ \Delta}$$

Diese Regel ist analog auch auf $[\cdot]$ anwendbar.

- if-then-else rechts (R20)

$$\frac{\Gamma,\ \epsilon\ \vdash\ \langle\alpha\rangle\varphi,\ \Delta \qquad \Gamma,\ \neg\epsilon\ \vdash\ \langle\beta\rangle\varphi,\ \Delta}{\Gamma\ \vdash\ \langle\texttt{IF}\ \epsilon\ \texttt{THEN}\ \alpha\ \texttt{ELSE}\ \beta\rangle\varphi,\ \Delta}$$

Diese Regel ist analog auch auf $[\cdot]$ anwendbar.

- if positive rechts (R21)

$$\frac{\Gamma\ \vdash\ \epsilon,\ \Delta \qquad \Gamma\ \vdash\ \langle\alpha\rangle\varphi,\ \Delta}{\Gamma\ \vdash\ \langle\texttt{IF}\ \epsilon\ \texttt{THEN}\ \alpha\ \texttt{ELSE}\ \beta\rangle\varphi,\ \Delta}$$

Diese Regel ist analog auch auf $[\cdot]$ anwendbar.

- if negative rechts (R22)

$$\frac{\Gamma\ \vdash\ \neg\epsilon,\ \Delta \qquad \Gamma\ \vdash\ \langle\beta\rangle\varphi,\ \Delta}{\Gamma\ \vdash\ \langle\texttt{IF}\ \epsilon\ \texttt{THEN}\ \alpha\ \texttt{ELSE}\ \beta\rangle\varphi,\ \Delta}$$

Diese Regel ist analog auch auf $[\cdot]$ anwendbar.

- if-then-else links (R23)

$$\frac{\Gamma,\ \epsilon,\ \langle\alpha\rangle\varphi\ \vdash\ \Delta \qquad \Gamma,\ \neg\epsilon,\ \langle\beta\rangle\varphi\ \vdash\ \Delta}{\Gamma,\ \langle\texttt{IF}\ \epsilon\ \texttt{THEN}\ \alpha\ \texttt{ELSE}\ \beta\rangle\varphi\ \vdash\ \Delta}$$

Diese Regel ist analog auch auf $[\cdot]$ anwendbar.

- if positive links (R24)

$$\frac{\Gamma\ \vdash\ \epsilon,\ \Delta \qquad \Gamma,\ \langle\alpha\rangle\varphi\ \vdash\ \Delta}{\Gamma,\ \langle\texttt{IF}\ \epsilon\ \texttt{THEN}\ \alpha\ \texttt{ELSE}\ \beta\rangle\varphi\ \vdash\ \Delta}$$

Diese Regel ist analog auch auf $[\cdot]$ anwendbar.

- if negative links (R25)

$$\frac{\Gamma\ \vdash\ \neg\epsilon,\ \Delta \qquad \Gamma,\ \langle\beta\rangle\varphi\ \vdash\ \Delta}{\Gamma,\ \langle\texttt{IF}\ \epsilon\ \texttt{THEN}\ \alpha\ \texttt{ELSE}\ \beta\rangle\varphi\ \vdash\ \Delta}$$

Diese Regel ist analog auch auf $[\cdot]$ anwendbar.

- while rechts (Invarianz) (R26)

$$\frac{\Gamma \ \vdash \ Inv, \ \Delta \qquad Inv, \ \epsilon \ \vdash \ [\alpha]Inv \qquad Inv, \ \neg\epsilon \ \vdash \ \varphi}{\Gamma \ \vdash \ [\texttt{WHILE} \ \epsilon \ \texttt{DO} \ \alpha]\varphi, \ \Delta}$$

  Inv ist eine beliebige Formel.

- while rechts (unwind) (R27)

$$\frac{\Gamma \ \vdash \ \epsilon, \ \Delta \qquad \Gamma \ \vdash \ \langle\alpha\rangle\langle\texttt{WHILE} \ \epsilon \ \texttt{DO} \ \alpha\rangle\varphi, \ \Delta}{\Gamma \ \vdash \ \langle\texttt{WHILE} \ \epsilon \ \texttt{DO} \ \alpha\rangle\varphi, \ \Delta}$$

  Diese Regel ist analog auch auf $[\cdot]$ anwendbar.

- while exit rechts (R28)

$$\frac{\Gamma \ \vdash \ \neg\epsilon, \ \Delta \qquad \Gamma \ \vdash \ \varphi, \ \Delta}{\Gamma \ \vdash \ \langle\texttt{WHILE} \ \epsilon \ \texttt{DO} \ \alpha\rangle\varphi, \ \Delta}$$

  Diese Regel ist analog auch auf $[\cdot]$ anwendbar.

- while links (unwind) (R29)

$$\frac{\Gamma \ \vdash \ \epsilon, \ \Delta \qquad \Gamma, \ \langle\alpha\rangle\langle\texttt{WHILE} \ \epsilon \ \texttt{DO} \ \alpha\rangle\varphi \ \vdash \ \Delta}{\langle\texttt{WHILE} \ \epsilon \ \texttt{DO} \ \alpha\rangle\varphi, \ \Gamma \ \vdash \ \Delta}$$

  Diese Regel ist analog auch auf $[\cdot]$ anwendbar.

- while exit links (R30)

$$\frac{\Gamma \ \vdash \ \neg\epsilon, \ \Delta \qquad \Gamma, \ \varphi \ \vdash \ \Delta}{\Gamma, \ \langle\texttt{WHILE} \ \epsilon \ \texttt{DO} \ \alpha\rangle\varphi \ \vdash \ \Delta}$$

  Diese Regel ist analog auch auf $[\cdot]$ anwendbar.

- $\forall$ links (R31)

$$\frac{\Gamma, \ \ [\alpha]\varphi, \ \forall\vec{x}\varphi \ \vdash \ \Delta}{\forall\vec{x}\varphi, \ \Gamma \ \vdash \ \Delta}$$

  Damit diese Regel anwendbar ist, müssen alle Variablen, die in dem Programm α auftauchen, in dem Vektor von Variablen $\vec{x}$ enthalten sein.

- $\exists$ rechts (R32)

$$\frac{\Gamma \ \vdash \ \langle\alpha\rangle\varphi, \ \exists\vec{x}\varphi, \ \Delta}{\Gamma \ \vdash \ \exists\vec{x}\varphi, \ \Delta}$$

  Damit diese Regel anwendbar ist, müssen alle Variablen, die in dem Programm α auftauchen, in dem Vektor von Variablen $\vec{x}$ enthalten sein.

- Generalisierung (R33)
$$\frac{\varphi \;\vdash\; \delta}{\langle\alpha\rangle\varphi \;\vdash\; \langle\varphi\rangle\delta}$$
Diese Regel ist analog auch auf $[\cdot]$ anwendbar.

7.2.2 Regeln für $\subset \cdot \supset$

Zur Erinnerung sei noch einmal anschaulich die Bedeutung der Modalität $\subset \cdot \supset$ wiederholt: eine gewisse Formel gelte vor, während und nach der Ausführung des Programms. Die Terminierung wird durch diese Modalität nicht impliziert.

- Zuweisung rechts (R34)
$$\frac{\Gamma \;\vdash\; \varphi,\;\Delta \qquad \Gamma_x^{x'},\; x = t_x^{x'} \;\vdash\; \varphi,\; \Delta_x^{x'}}{\Gamma \;\vdash\; \subset x := t \supset \varphi,\;\Delta}$$
Da die Zuweisung ein atomares Programmkonstrukt ist, wird angenommen, daß sich während der Ausführung einer Zuweisung der Zustand nicht ändert. Die Bedeutung der Modalität $\subset \cdot \supset$ reduziert sich bei der Zuweisung also darauf, daß eine Formel vorher und nachher erfüllt ist. Dies wird durch die beiden Prämissen ausgedrückt.

- Zuweisung links (R35)
$$\frac{\varphi_x^{x'},\;\varphi,\; x = t_x^{x'},\; \Gamma_x^{x'} \;\vdash\; \Delta_x^{x'}}{\subset x := t \supset \varphi,\; \Gamma \;\vdash\; \Delta}$$

- ; rechts (R36)
$$\frac{\Gamma \;\vdash\; \subset\alpha\supset\varphi,\;\Delta \qquad \Gamma \;\vdash\; [\alpha]\subset\beta\supset\varphi,\;\Delta}{\Gamma \;\vdash\; \subset\alpha;\beta\supset\varphi,\;\Delta}$$
Die Prämissen dieser Regeln bedeuten, daß φ einerseits vor, während und nach der Ausführung von α gelten muß und andererseits auch vor, während und nach der Ausführung von β, gestartet in den Endzuständen von α (im deterministischen Fall ist dies höchstens einer, die Regel wäre aber auch im indeterministischen Fall anwendbar).

- ; links (R37)
$$\frac{\Gamma,\; \subset\alpha\supset\varphi,\; [\alpha]\subset\beta\supset\varphi \;\vdash\; \Delta}{\Gamma,\; \subset\alpha;\beta\supset\varphi \;\vdash\; \Delta}$$

- if-then-else rechts (R38)

$$\frac{\Gamma,\ \epsilon\ \vdash\ \subset\alpha\supset\varphi,\ \Delta \qquad \Gamma,\ \neg\epsilon\ \vdash\ \subset\beta\supset\varphi,\ \Delta}{\Gamma\ \vdash\ \subset\texttt{IF}\ \epsilon\ \texttt{THEN}\ \alpha\ \texttt{ELSE}\ \beta\supset\varphi,\ \Delta}$$

- if positive rechts (R39)

$$\frac{\Gamma\ \vdash\ \epsilon,\ \Delta \qquad \Gamma\ \vdash\ \subset\alpha\supset\varphi,\ \Delta}{\Gamma\ \vdash\ \subset\texttt{IF}\ \epsilon\ \texttt{THEN}\ \alpha\ \texttt{ELSE}\ \beta\supset\varphi,\ \Delta}$$

- if negative rechts (R40)

$$\frac{\Gamma\ \vdash\ \neg\epsilon,\ \Delta \qquad \Gamma\ \vdash\ \subset\beta\supset\varphi,\ \Delta}{\Gamma\ \vdash\ \subset\texttt{IF}\ \epsilon\ \texttt{THEN}\ \alpha\ \texttt{ELSE}\ \beta\supset\varphi,\ \Delta}$$

- if-then-else links (R41)

$$\frac{\Gamma,\ \epsilon,\ \subset\alpha\supset\varphi\ \vdash\ \Delta \qquad \Gamma,\ \neg\epsilon,\ \subset\beta\supset\varphi\ \vdash\ \Delta}{\Gamma,\ \subset\texttt{IF}\ \epsilon\ \texttt{THEN}\ \alpha\ \texttt{ELSE}\ \beta\supset\varphi\ \vdash\ \Delta}$$

- if positive links (R42)

$$\frac{\Gamma\ \vdash\ \epsilon,\ \Delta \qquad \Gamma,\ \subset\alpha\supset\varphi\ \vdash\ \Delta}{\Gamma,\ \subset\texttt{IF}\ \epsilon\ \texttt{THEN}\ \alpha\ \texttt{ELSE}\ \beta\supset\varphi\ \vdash\ \Delta}$$

- if negative links (R43)

$$\frac{\Gamma\ \vdash\ \neg\epsilon,\ \Delta \qquad \Gamma,\ \subset\beta\supset\varphi\ \vdash\ \Delta}{\Gamma,\ \subset\texttt{IF}\ \epsilon\ \texttt{THEN}\ \alpha\ \texttt{ELSE}\ \beta\supset\varphi\ \vdash\ \Delta}$$

- while rechts (Invarianz) (R44)

$$\frac{\Gamma \vdash Inv,\ \Delta \quad Inv,\ \epsilon \vdash [\alpha]Inv \quad Inv,\ \epsilon \vdash \subset\alpha\supset\varphi \quad Inv,\ \neg\epsilon \vdash \varphi}{\Gamma\ \vdash\ \subset\texttt{WHILE}\ \epsilon\ \texttt{DO}\ \alpha\supset\varphi,\ \Delta}$$

  Inv ist eine beliebige Formel.
  Durch die Prämissen der Regel wird sichergestellt, daß die Formel Inv eine Invariante der Schleife ist, d.h. wenn die Schleifenbedingung erfüllt ist und Inv gilt, dann gilt Inv auch nach der Ausführung des Schleifenrumpfes. In einigen Fällen mag φ als Inv genügen, doch im allgemeinen wird φ nicht stark genug sein.

- while links (unwind) (R45)

$$\frac{\Gamma \ \vdash \ \epsilon, \ \Delta \qquad \Gamma, \ [\alpha]\subset \texttt{WHILE} \ \epsilon \ \texttt{DO} \ \alpha\supset\varphi \ \vdash \ \Delta}{\Gamma, \ \subset \texttt{WHILE} \ \epsilon \ \texttt{DO} \ \alpha\supset\varphi \ \vdash \ \Delta}$$

- while exit links (R46)

$$\frac{\Gamma, \ \neg\epsilon, \ \varphi \ \vdash \ \Delta \qquad \Gamma, \ \subset\alpha\supset\varphi \ \vdash \ \Delta}{\Gamma, \ \subset \texttt{WHILE} \ \epsilon \ \texttt{DO} \ \alpha\supset\varphi \ \vdash \ \Delta}$$

- $\forall$ links (R47)

$$\frac{\Gamma, \quad \subset\alpha\supset\varphi, \ \forall\vec{x}\varphi \ \vdash \ \Delta}{\forall\vec{x}\varphi, \ \Gamma \ \vdash \ \Delta}$$

  Damit diese Regel anwendbar ist, müssen alle Variablen, die in dem Programm α auftauchen, in dem Vektor von Variablen $\vec{x}$ enthalten sein.

- Generalisierung (R48)

$$\frac{\varphi \ \vdash \ \delta}{\subset\alpha\supset\varphi \ \vdash \ \subset\varphi\supset\delta}$$

7.2.3 Regeln für $\|\cdot\|$

Anschaulich gesprochen, drückt die Modalität $\|\cdot\|$ aus, daß die Gültigkeit einer Formel während der Programmausführung erhalten bleibt, d.h., wird diese Formel während der Programmausführung einmal wahr, dann bleibt sie während der restlichen Ausführung wahr (insbesondere gilt die Formel dann auch im Endzustand, also nach der Ausführung). Die Terminierung wird durch diese Modalität nicht garantiert.

- Zuweisung rechts (R49)

$$\frac{\Gamma \ \vdash \ \neg\varphi, \ [x := t]\varphi, \ \Delta}{\Gamma \ \vdash \ \|x := t\|\varphi, \ \Delta}$$

  Diese Regel nutzt die Tatsache aus, daß Zuweisungen als atomare Programme keine Zwischenzustände zwischen Anfangs- und Endzustand haben. Gilt φ im Anfangszustand nicht, dann ist aus diesem Grund egal, ob φ im Endzustand gilt oder nicht.

- Zuweisung links (R50)

$$\frac{\neg\varphi,\ \Gamma\ \vdash\ \Delta \qquad \subset x := t\supset\varphi,\ \Gamma\ \vdash\ \Delta}{\|x := t\|\varphi,\ \Gamma\ \vdash\ \Delta}$$

- ; rechts (R51)

$$\frac{\Gamma\ \vdash\ \|\alpha\|\varphi,\ \Delta \qquad \Gamma\ \vdash\ [\alpha]\|\beta\|\varphi,\ \Delta}{\Gamma\ \vdash\ \|\alpha;\beta\|\varphi,\ \Delta}$$

Diese Regel scheint zuerst nur für den Fall, daß φ während der Ausführung von α nicht wahr wurde, einsichtig. Im anderen Fall ist nach Definition von $\|\cdot\|$ φ im Endzustand der Ausführung von α wahr. Nach Definition der Verkettung von Programmen fällt dieser Endzustand aber mit dem Anfangszustand von β zusammen, d.h. φ gilt zu Beginn der Ausführung von β und somit ist die Regeln korrekt.

- ; links (R52)

$$\frac{\Gamma,\ \|\alpha\|\varphi,\ [\alpha]\|\beta\|\varphi\ \vdash\ \Delta}{\Gamma,\ \|\alpha;\beta\|\varphi\ \vdash\ \Delta}$$

- if-then-else rechts (R53)

$$\frac{\Gamma,\ \epsilon\ \vdash\ \|\alpha\|\varphi,\ \Delta \qquad \Gamma,\ \neg\epsilon\ \vdash\ \|\beta\|\varphi,\ \Delta}{\Gamma\ \vdash\ \|\mathtt{IF}\ \epsilon\ \mathtt{THEN}\ \alpha\ \mathtt{ELSE}\ \beta\|\varphi,\ \Delta}$$

- if positive rechts (R54)

$$\frac{\Gamma\ \vdash\ \epsilon,\ \Delta \qquad \Gamma\ \vdash\ \|\alpha\|\varphi,\ \Delta}{\Gamma\ \vdash\ \|\mathtt{IF}\ \epsilon\ \mathtt{THEN}\ \alpha\ \mathtt{ELSE}\ \beta\|\varphi,\ \Delta}$$

- if negative rechts (R55)

$$\frac{\Gamma\ \vdash\ \neg\epsilon,\ \Delta \qquad \Gamma\ \vdash\ \|\beta\|\varphi,\ \Delta}{\Gamma\ \vdash\ \|\mathtt{IF}\ \epsilon\ \mathtt{THEN}\ \alpha\ \mathtt{ELSE}\ \beta\|\varphi,\ \Delta}$$

- if-then-else links (R56)

$$\frac{\Gamma,\ \epsilon,\ \|\alpha\|\varphi\ \vdash\ \Delta \qquad \Gamma,\ \neg\epsilon,\ \|\beta\|\varphi\ \vdash\ \Delta}{\Gamma,\ \|\mathtt{IF}\ \epsilon\ \mathtt{THEN}\ \alpha\ \mathtt{ELSE}\ \beta\|\varphi\ \vdash\ \Delta}$$

- if positive links (R57)

$$\frac{\Gamma \ \vdash \ \epsilon, \ \Delta \qquad \Gamma, \ \|\alpha\|\varphi \ \vdash \ \Delta}{\Gamma, \ \|\texttt{IF} \ \epsilon \ \texttt{THEN} \ \alpha \ \texttt{ELSE} \ \beta\|\varphi \ \vdash \ \Delta}$$

- if negative links (R58)

$$\frac{\Gamma \ \vdash \ \neg\epsilon, \ \Delta \qquad \Gamma, \ \|\beta\|\varphi \ \vdash \ \Delta}{\Gamma, \ \|\texttt{IF} \ \epsilon \ \texttt{THEN} \ \alpha \ \texttt{ELSE} \ \beta\|\varphi \ \vdash \ \Delta}$$

- while rechts (unwind) (R59)

$$\frac{\Gamma \ \vdash \ \epsilon, \ \Delta \qquad \Gamma \ \vdash \ [\alpha]\|\texttt{WHILE} \ \epsilon \ \texttt{DO} \ \alpha\|\varphi, \ \Delta \qquad \Gamma \ \vdash \ \|\alpha\|\varphi, \ \Delta}{\Gamma \ \vdash \ \|\texttt{WHILE} \ \epsilon \ \texttt{DO} \ \alpha\|\varphi, \ \Delta}$$

  Diese Regel wickelt die Schleife analog der Regel für $\subset \cdot \supset$ ab.

- while exit rechts (R60)

$$\frac{\Gamma \ \vdash \ \neg\epsilon, \ \Delta}{\Gamma \ \vdash \ \|\texttt{WHILE} \ \epsilon \ \texttt{DO} \ \alpha\|\varphi, \ \Delta}$$

- while links (unwind) (R61)

$$\frac{\Gamma \ \vdash \ \epsilon, \ \Delta \qquad \Gamma, \ [\alpha]\|\texttt{WHILE} \ \epsilon \ \texttt{DO} \ \alpha\|\varphi, \ \|\alpha\|\varphi \ \vdash \ \Delta}{\Gamma, \ \|\texttt{WHILE} \ \epsilon \ \texttt{DO} \ \alpha\|\varphi \ \vdash \ \Delta}$$

- while exit links (R62)

$$\frac{\Gamma \ \vdash \ \neg\epsilon, \ \Delta \qquad \Gamma \ \vdash \ \Delta}{\Gamma, \ \|\texttt{WHILE} \ \epsilon \ \texttt{DO} \ \alpha\|\varphi \ \vdash \ \Delta}$$

- $\forall$ links (R63)

$$\frac{\Gamma, \quad \|\alpha\|\varphi, \ \forall\vec{x}\varphi \ \vdash \ \Delta}{\forall\vec{x}\varphi, \ \Gamma \ \vdash \ \Delta}$$

  Damit diese Regel anwendbar ist, müssen alle Variablen, die in dem Programm α auftauchen, in dem Vektor von Variablen $\vec{x}$ enthalten sein.

- $\exists$ rechts (R64)

$$\frac{\Gamma \ \vdash \ \|\alpha\|\varphi, \ \exists\vec{x}\varphi, \ \Delta}{\Gamma \ \vdash \ \exists\vec{x}\varphi, \ \Delta}$$

  Damit diese Regel anwendbar ist, müssen alle Variablen, die in dem Programm α auftauchen, in dem Vektor von Variablen $\vec{x}$ enthalten sein.

- Generalisierung (R65)

$$\frac{\varphi \;\vdash\; \delta}{\|\alpha\|\varphi \;\vdash\; \|\varphi\|\delta}$$

7.2.4 Regeln für $\ll \cdot \gg$

Die Modalität $\ll \cdot \gg$ drückt aus, daß es einen terminierenden Programmablauf gibt, während dessen eine gewisse Formel mindestens einmal wahr wird (Anfangs- und Endzustand eingeschlossen).

- Zuweisung rechts (R66)

$$\frac{\Gamma_x^{x'},\; x = t_x^{x'} \;\vdash\; \varphi_x^{x'},\; \varphi,\; \Delta_x^{x'}}{\Gamma \;\vdash\; \ll x := t \gg \varphi,\; \Delta}$$

Die Prämisse dieser Regel drückt aus, daß φ entweder schon vor der Zuweisung gilt oder hinterher oder beides.

- Zuweisung links (R67)

$$\frac{\Gamma,\; \varphi \;\vdash\; \Delta \qquad x = t_x^{x'},\; \Gamma_x^{x'} \;\vdash\; \Delta_x^{x'}}{\ll x := t \gg \varphi,\; \Gamma \;\vdash\; \Delta}$$

- ; rechts (R68)

$$\frac{\Gamma \;\vdash\; \ll \alpha \gg \varphi,\; \langle\alpha\rangle \ll \beta \gg \varphi,\; \Delta}{\Gamma \;\vdash\; \ll \alpha;\beta \gg \varphi,\; \Delta}$$

Bemerkung:
Diese Regel wäre in [Prat79] nicht korrekt, da dort eine andere Semantik für die Modalität $\ll \cdot \gg$ definiert wird, die die Terminierung nicht garantiert und die ausdrückt, daß während jedes möglichen Programmablaufs eine Formel mindestens einmal erfüllt sein muß. Hätte man die Semantik von Pratt übernommen, wäre diese Regel nur für deterministische Programme korrekt. Ferner wären die Modalitäten $\ll \cdot \gg$ und $\subset \cdot \supset$ dann auch nicht mehr komplementär zueinander.

- ; links (R69)

$$\frac{\Gamma,\; \ll \alpha \gg \varphi \;\vdash\; \Delta \qquad \Gamma,\; \langle\alpha\rangle \ll \beta \gg \varphi \;\vdash\; \Delta}{\Gamma,\; \ll \alpha;\beta \gg \varphi \;\vdash\; \Delta}$$

- if-then-else rechts (R70)

$$\frac{\Gamma,\ \epsilon\ \vdash\ \ll\alpha\gg\varphi,\ \Delta \qquad \Gamma,\ \neg\epsilon\ \vdash\ \ll\beta\gg\varphi,\ \Delta}{\Gamma\ \vdash\ \ll\mathtt{IF}\ \epsilon\ \mathtt{THEN}\ \alpha\ \mathtt{ELSE}\ \beta\gg\varphi,\ \Delta}$$

- if positive rechts (R71)

$$\frac{\Gamma\ \vdash\ \epsilon,\ \Delta \qquad \Gamma\ \vdash\ \ll\alpha\gg\varphi,\ \Delta}{\Gamma\ \vdash\ \ll\mathtt{IF}\ \epsilon\ \mathtt{THEN}\ \alpha\ \mathtt{ELSE}\ \beta\gg\varphi,\ \Delta}$$

- if negative rechts (R72)

$$\frac{\Gamma\ \vdash\ \neg\epsilon,\ \Delta \qquad \Gamma\ \vdash\ \ll\beta\gg\varphi,\ \Delta}{\Gamma\ \vdash\ \ll\mathtt{IF}\ \epsilon\ \mathtt{THEN}\ \alpha\ \mathtt{ELSE}\ \beta\gg\varphi,\ \Delta}$$

- if-then-else links (R73)

$$\frac{\Gamma,\ \epsilon,\ \ll\alpha\gg\varphi\ \vdash\ \Delta \qquad \Gamma,\ \neg\epsilon,\ \ll\beta\gg\varphi\ \vdash\ \Delta}{\Gamma,\ \ll\mathtt{IF}\ \epsilon\ \mathtt{THEN}\ \alpha\ \mathtt{ELSE}\ \beta\gg\varphi\ \vdash\ \Delta}$$

- if positive links (R74)

$$\frac{\Gamma\ \vdash\ \epsilon,\ \Delta \qquad \Gamma,\ \ll\alpha\gg\varphi\ \vdash\ \Delta}{\Gamma,\ \ll\mathtt{IF}\ \epsilon\ \mathtt{THEN}\ \alpha\ \mathtt{ELSE}\ \beta\gg\varphi\ \vdash\ \Delta}$$

- if negative links (R75)

$$\frac{\Gamma\ \vdash\ \neg\epsilon,\ \Delta \qquad \Gamma,\ \ll\beta\gg\varphi\ \vdash\ \Delta}{\Gamma,\ \ll\mathtt{IF}\ \epsilon\ \mathtt{THEN}\ \alpha\ \mathtt{ELSE}\ \beta\gg\varphi\ \vdash\ \Delta}$$

- while exit rechts (R76)

$$\frac{\Gamma,\ \neg\epsilon\ \vdash\ \varphi,\ \Delta \qquad \Gamma,\ \epsilon\ \vdash\ \ll\alpha\gg\varphi,\ \Delta}{\Gamma\ \vdash\ \ll\mathtt{WHILE}\ \epsilon\ \mathtt{DO}\ \alpha\gg\varphi,\ \Delta}$$

  Diese Regel ist nur für deterministische Programme gültig.

- while links (Invarianz) (R77)

$$\frac{\Gamma \vdash Inv,\ \Delta \quad Inv,\ \epsilon \vdash [\alpha]Inv \quad Inv,\ \epsilon,\ \ll\alpha\gg\varphi \vdash \quad Inv,\ \neg\epsilon,\ \varphi \vdash}{\Gamma,\ \ll\mathtt{WHILE}\ \epsilon\ \mathtt{DO}\ \alpha\gg\varphi\ \vdash\ \Delta}$$

  Diese Regel ist nur für deterministische Programme gültig.

- while rechts (unwind) (R78)

$$\frac{\Gamma \;\vdash\; \epsilon,\; \Delta \qquad \Gamma \;\vdash\; \langle\alpha\rangle \ll\texttt{WHILE}\ \epsilon\ \texttt{DO}\ \alpha\gg\varphi,\; \Delta}{\Gamma \;\vdash\; \ll\texttt{WHILE}\ \epsilon\ \texttt{DO}\ \alpha\gg\varphi,\; \Delta}$$

  Diese Regel ist nur für deterministische Programme gültig.

- $\exists$ rechts (R79)

$$\frac{\Gamma \;\vdash\; \ll\alpha\gg\varphi,\; \exists\vec{x}\varphi,\; \Delta}{\Gamma \;\vdash\; \exists\vec{x}\varphi,\; \Delta}$$

  Damit diese Regel anwendbar ist, müssen alle Variablen, die in dem Programm α auftauchen, in dem Vektor von Variablen $\vec{x}$ enthalten sein.

- Generalisierung (R80)

$$\frac{\varphi \;\vdash\; \delta}{\ll\alpha\gg\varphi \;\vdash\; \ll\varphi\gg\delta}$$

7.3 Korrektheit

In diesem Abschnitt soll die Korrektheit des Kalküls gezeigt werden. Die Beweise sind nicht sehr anspruchsvoll aber oft sehr „technisch“, vor allem die Korrektheitsbeweise der Zuweisungsregeln.
Zuerst werden aber noch einige für die Beweise sehr hilfreiche Lemmata angegeben. Das erste Lemma ist ein Substitutionslemma für die Prozeßlogik. Dieses wird benötigt, um die beiden anderen Lemmata zu beweisen, die dann in den Korrektheitsbeweisen hauptsächlich benötigt werden.

Lemma 39.
Sei eine Signatur Σ gegeben und sei $\alpha \in \Pi$, σ eine Variablenumbenennung für α , $\mathcal{M}$ ein Struktur und seien $S = \{s_0, s_1, \ldots, s_n\}, \hat{S} = \{\hat{s}_0, \hat{s}_1, \ldots, \hat{s}_n\}$ Zustandsfolgen mit $\hat{s}_i(y) := I(\mathcal{M}, s_i)(\sigma(y)) \quad (0 \leq i \leq n)$ für alle Variablen y. Dann gilt:

$$\hat{S} \in \phi(\alpha) \text{ gdw. } S \in \phi(\sigma(\alpha)).$$

Beweis:
Der Beweis wird durch strukturelle Induktion über den Programmaufbau geführt.

Seien $S = \{s_0, s_1, \ldots, s_n\}, \hat{S} = \{\hat{s}_0, \hat{s}_1, \ldots, \hat{s}_n\}$ Zustandsfolgen mit $\hat{s}_i(y) := I(\mathcal{M}, s_i)(\sigma(y))$, $\alpha, \beta, \gamma \in \Pi$ und $F \in Fml_{pl}$, F programmfrei, d.h. F rein prädikatenlogisch.
Induktionsanfang:

- $\alpha = (y := t)$:
  $(s_0, s_1) \in \phi(\sigma(y := t))$ *gdw.*
  $s_0 \neq \Lambda,\ s_1 \neq \Lambda,\ s_1(\sigma(z)) = s_0(\sigma(z))$ *für* $\sigma(z) \neq \sigma(y)$ *und* $s_1(\sigma(y)) = I(\mathcal{M}, s_0)(\sigma(t))$
  gdw. * $\hat{s} \neq \Lambda,\ \hat{s}_1 \neq \Lambda,\ \hat{s}_1(z) = \hat{s}_0(z)$ *für* $z \neq y$ *und* $\hat{s}_1(y) = I(\mathcal{M}, \hat{s}_0)(t)$
  gdw. $(\hat{s}_0, \hat{s}_1) \in \phi(y := t)$
  * gilt wegen:
  $\hat{s}_0(z) = s_0(\sigma(z)) = s_1(\sigma(z)) = \hat{s}_1(z)$ *für* $z \neq y$
  ($z \neq y$ *impliziert* $\sigma(z) \neq \sigma(y)$ *(Injektivität von* σ))
  $\hat{s}_1(y) = s(\sigma(y)) = I(\mathcal{M}, s_0)(\sigma(t)) = I(\mathcal{M}, \hat{s}_0)(t)$
  und umgekehrt
  $s_0(\sigma(z)) = \hat{s}_0(z) = \hat{s}_1(z) = s_1(\sigma(z))$ *für* $\sigma(z) \neq \sigma(y)$
  ($\sigma(z) \neq \sigma(y)$ *impliziert* $z \neq y$)
  $s(\sigma(y)) = \hat{s}_1(y) = I(\mathcal{M}, \hat{s}_0)(t) = I(\mathcal{M}, s_0)(\sigma(t))$

- $\alpha = (F?)$:
  $s \in \phi(\sigma(F?))$ *gdw.* $s \in \phi(\sigma(F)?)$ *gdw.* $s \in \tau(\sigma(F))$
  gdw. (Substitutionslemma der Prädikatenlogik) $\hat{s} \in \tau(F)$ *gdw.* $\hat{s} \in \phi(F?)$

Induktionsschluß:

- $\alpha = (\beta; \gamma)$:
  $(s_1, s_2, \dots, s_n) \in \phi(\sigma(\beta; \gamma))$ *gdw.*
  $(s_1, s_2, \dots, s_n) \in \phi(\sigma(\beta)); \phi(\sigma(\gamma))$ *gdw.*
  $(r_1, r_2, \dots, r_l) \in \phi(\sigma(\beta))$ *und* $(t_1, t_2, \dots, t_m) \in \phi(\sigma(\gamma))$ *und*
  $(s_1, s_2, \dots, s_n) = (r_1, r_2, \dots, r_l); (t_1, t_2, \dots, t_m)$ *gdw. (Induktionsannahme)*
  $(\hat{r}_1, \hat{r}_2, \dots, \hat{r}_l) \in \phi(\beta)$ *und* $(\hat{t}_1, \hat{t}_2, \dots, \hat{t}_m) \in \phi(\gamma)$ *und*
  $(\hat{s}_1, \hat{s}_2, \dots, \hat{s}_n) = (\hat{r}_1, \hat{r}_2, \dots, \hat{r}_l); (\hat{t}_1, \hat{t}_2, \dots, \hat{t}_m)$ *gdw.*
  $(\hat{s}_1, \hat{s}_2, \dots, \hat{s}_n) \in \phi(\beta); \phi(\gamma)$ *gdw.*
  $(\hat{s}_1, \hat{s}_2, \dots, \hat{s}_n) \in \phi(\beta; \gamma)$

- $\alpha = (\beta^*)$: *analog zu* $(\beta; \gamma)$,

- $\alpha = (\beta \cup \gamma)$: *analog zu* $(\beta; \gamma)$

■

Lemma 40 (Substitutionslemma).
Sei $\varphi \in Fml_{pl}$, σ *eine kollisionsfreie Substitution für* φ , $\mathcal{M}$ *eine Struktur und seien* $\hat{s}, s$ *Zustände mit* $\hat{s}(y) := I(\mathcal{M}, s)(\sigma(y))$ *für alle Variablen* y. *Dann gilt:*

$$\hat{s} \in \tau(\varphi) \text{ gdw. } s \in \tau(\sigma(\varphi)).$$

Beweis:
Der Beweis wird durch strukturelle Induktion über den Formelaufbau geführt.
Sei $s \in \tau(\sigma(\varphi))$ beliebig, $\alpha, \beta, \gamma \in \Pi$, $\varphi, F_1, F_2 \in Fml_{pl}$.
Induktionsanfang:

- $\varphi = true, \varphi = false$: klar,

- φ atomar, d.h. $\varphi = R(t_1, \ldots, t_n)$:
  $s \in \tau(\sigma(R(t_1, \ldots, t_n)))$ *gdw.* $val_{\mathcal{M},s}\sigma(R(t_1, \ldots, t_n)) = true$
  gdw.* $val_{\mathcal{M},\hat{s}}(R(t_1, \ldots, t_n)) = true$ *gdw.* $\hat{s} \in \tau(R(t_1, \ldots, t_n))$,
  * gilt nach dem Substitutionslemma der Prädikatenlogik.

Induktionsschluß:

- $\varphi = F_1 \vee F_2$:
  $s \in \tau(\sigma(F_1 \vee F_2))$ *gdw.* $s \in \tau(\sigma(F_1)) \cup \tau(\sigma(F_2))$
  gdw. $s \in \tau(\sigma(F_1))$ *oder* $s \in \tau(\sigma(F_2))$
  gdw. (Induktionsannahme) $\hat{s} \in \tau(F_1)$ *oder* $\hat{s} \in \tau(F_2)$
  gdw. $\hat{s} \in \tau(F_1) \cup \tau(F_2)$ *gdw.* $\hat{s} \in \tau(F_1 \vee F_2)$,

- $\varphi = \exists x F$:
  $s \in \tau(\sigma(\exists x F))$ *gdw.* $s \in \tau(\exists x \sigma_x(F))$ *gdw.*
  es existiert ein $d \in W$ *mit* $s_x^d \in \tau(\sigma_x(F))$ *gdw. (Induktionsannahme**)*
  es existiert ein $d \in W$ *mit* $\hat{s}_x^d \in \tau(F)$ *gdw.*
  $\hat{s} \in \tau(\exists x F))$
  ** gilt, wenn $\hat{s}_x^d(y) = s_x^d(\sigma_x(y))$ für alle Variablen y :
  sei $y = x : \hat{s}_x^d(x) = d = s_x^d(x) = s_x^d(\sigma_x(x))$
  sei $y \neq x : \hat{s}_x^d(y) = \hat{s}(y) = s(\sigma(y)) = s_x^d(\sigma(y)) = s_x^d(\sigma_x(y))$
  Die Gleichheit $s(\sigma(y)) = s_x^d(\sigma(y))$ wird durch die Kollisionsfreiheit von σ garantiert.

- $\varphi = \neg F$:
  $s \in \tau(\sigma(\neg F))$ *gdw.* $s \in \tau(\neg(\sigma(F)))$ *gdw.* $s \in W \setminus \tau(\sigma(F))$ *gdw.* $s \notin \tau(\sigma(F))$
  gdw. (Induktionsannahme) $\hat{s} \notin \tau(F)$ *gdw.* $\hat{s} \in W \setminus \tau(F)$ *gdw.* $\hat{s} \in \tau(\neg F)$,

- $\varphi = \subset\alpha\supset F$:
  $s \in \tau(\sigma(\subset\alpha\supset F))$ *gdw.* $s \in \tau(\sigma(\subset\alpha\supset)\sigma(F))$
  gdw. für alle $S \in \phi(\sigma(\alpha))$ *mit* $S = (s, s_1, \ldots s_{m_S})$ *gilt:*
  $\forall s_i : s_i \neq \Lambda \rightarrow s_i \in \tau(\sigma(F))$
  gdw. (Lemma 39) für alle $\hat{S} \in \phi(\alpha)$ *mit* $\hat{S} = (\hat{s}, \hat{s}_1, \ldots \hat{s}_{m_S})$ *gilt:*
  $\forall \hat{s}_i : \hat{s}_i \neq \Lambda \rightarrow \hat{s}_i \in \tau(F)$ *gdw.* $\hat{s} \in \tau(\subset\alpha\supset F)$,

- $\varphi = \|\alpha\| F$: *analog zu* $\subset\alpha\supset F$

- $\varphi = \langle\alpha\rangle F$: *analog zu* $\subset\alpha\supset F$
- $\varphi = \ll\alpha\gg F$: *analog zu* $\subset\alpha\supset F$,

■

Lemma 41.
Sei $\sigma = \{x \leftarrow u\}$ *eine Substitution mit* $x \notin var(\varphi)$, *sei* u *ein Term und seien* $s, \hat{s}$ *Zustände mit* $\hat{s}(y) := I(\mathcal{M}, s)(\sigma(y))$ *für alle Variablen* y. *Dann gilt*

$$s \in \tau(\varphi) \textit{ gdw. } \hat{s} \in \tau(\varphi).$$

Beweis:
Aus dem Substitutionslemma folgt $s \in \tau(\sigma(\varphi))$ *gdw.* $\hat{s} \in \tau(\varphi)$ und aus $dom(\sigma) \cap var(\varphi) = \emptyset$ folgt $\sigma(\varphi) = \varphi$.

■

Lemma 42.
Sei $\varphi \in Fml_{pl}$, x, x' *Variablen,* $\sigma = \{x \leftarrow x'\}$ *eine Substitution für* φ *mit* $x' \notin var(\varphi)$, $\sigma^{-1} = \{x' \leftarrow x\}$ *und seien* $\hat{s}, s$ *Zustände mit* $\hat{s}(y) := s(\sigma^{-1}(y))$ *für alle Variablen* y. *Dann gilt:*

$$s \in \tau(\varphi) \textit{ gdw. } \hat{s} \in \tau(\sigma(\varphi)).$$

Beweis:
Nach dem Substitutionslemma gilt $t \in \tau(\varphi)$ *gdw.* $\hat{s} \in \tau(\sigma(\varphi))$ mit $t(y) := \hat{s}(\sigma(y))$. Nach Lemma 41 gilt $t \in \tau(\varphi)$ *gdw.* $s \in \tau(\varphi)$, falls $t(y) = s(\sigma^{-1}(y))$. Es gilt $t(y) = \hat{s}(\sigma(y)) = s(\sigma^{-1}(\sigma(y))) = s(\sigma^{-1}(\sigma^{-1}(\sigma(y)))) = s(\sigma^{-1}(y))$.

■

Satz 43 (Korrektheit).
Der Kalkül, der aus den Regeln (R1)-(R79) besteht, ist korrekt, d.h. für jede Formelmenge $\Gamma \subseteq Fml_{pl}$ *und jede Formel* $\varphi \in Fml_{pl}$ *gilt:*

$$\Gamma \vdash \varphi \textit{ impliziert } \models \Gamma \to \varphi.$$

Beweisskizze:
Um diesen Satz zu beweisen, muß man die Korrektheit jeder einzelnen Regel beweisen. Dies geschieht dadurch, daß man annimmt, die Prämissen der Regel seien allgemeingültig. Daraus muß man folgern können, daß auch die mit der Regel abgeleitete Konklusion allgemeingültig ist. Im folgenden wird dies an zwei Regeln exemplarisch gezeigt.

7.3.1 Beweis der Korrektheit der Regel (Zuweisung rechts) (R34)

$$\frac{\Gamma \ \vdash \ \varphi, \ \Delta \qquad \Gamma_x^{x'}, \ x = t_x^{x'} \ \vdash \ \varphi, \ \Delta_x^{x'}}{\Gamma \ \vdash \ \subset x := t \supset \varphi, \ \Delta}$$

Lemma 44.
Die Regel (Zuweisung rechts) (R34) ist korrekt.

Beweis:
Nach Definition der Funktion τ ist

$$\begin{aligned}\tau(\subset x := t \supset \varphi) = \ \{s_0 \in \tau(\varphi) \ \mid \ & \textit{es gibt ein } s_1 \in W \textit{ mit} \\ & s_1(y) = I(\mathcal{M}, s_0)(\sigma(y)) \textit{ für alle Variablen } y, \\ & \sigma = \{x \leftarrow t\} \textit{ und } s_1 \in \tau(\varphi)\}.\end{aligned}$$

Im Beweis wird eine andere, aber nach Lemma 41 äquivalente Definition benötigt:

$$\begin{aligned}\tau(\subset x := t \supset \varphi) = \ \{s_0 \in \tau(\varphi) \ \mid \ & \textit{es gibt ein } s_1 \in W \textit{ mit} \\ & s_1(y) = I(\mathcal{M}, s_0)(\sigma(y)) \textit{ für alle Variablen } y \in var(\varphi), \\ & \sigma = \{x \leftarrow t\} \textit{ und } s_1 \in \tau(\varphi)\}.\end{aligned}$$

Im Beweis wird die in Abschnitt 4.2 definierte Schreibweise $(\mathcal{M}, s) \models F$ für $s \in \tau(F)$ verwendet. Der Übersichtlichkeit wegen wird die Struktur $\mathcal{M}$ weggelassen und so wird $s \models F$ für die Tatsache geschrieben, daß die Formeln F im Zustand s gültig ist.

Der Korrektheitsbeweis wird als Widerspruchsbeweis geführt, d.h. man nimmt, an die Prämissen der Regeln seien allgemeingültig, die Konklusion aber nicht.

Annahme:
Die Prämissen seien allgemeingültig, die Konklusion jedoch nicht. Es existiere also eine Struktur $\mathcal{M}$ und ein Zustand s mit $s \models \Gamma, s \not\models \subset x := t \supset \varphi, s \not\models \Delta$.

Fall 1: $s \not\models \varphi$
In diesem Fall wird die Sequenz $\Gamma \vdash \varphi,\ \ \Delta$ wegen $s \models \Gamma$ und $s \not\models \Delta$ zu *false* ausgewertet, was ein Widerspruch zur Annahme der Allgemeingültigkeit der Prämissen darstellt.

Fall 2: $s \models \varphi$
Definiere zwei Substitutionen

$$\begin{aligned} \sigma_1 &:= \{x' \leftarrow x\}, \\ \sigma_2 &:= \{x \leftarrow t_x^{x'}\} \end{aligned}$$

und zwei neue Zustände

$$\begin{aligned} &s_1(y) := I(\mathcal{M}, s)(\sigma_1(y)), \\ &s_2(y) = I(\mathcal{M}, s_1)(\sigma_2(y)) \textit{ für alle Variablen } y. \end{aligned}$$

x' sei die Variable, die bei der Regelanwendung verwendet wurde.
Damit gilt:
$s_2(x) = I(\mathcal{M}, s_1)(\sigma_2(x)) = I(\mathcal{M}, s_1)(t_x^{x'}) = I(\mathcal{M}, s)(\sigma_1(t_x^{x'})) = I(\mathcal{M}, s)(t)$
$I(\mathcal{M}, s_2)(t_x^{x'}) = I(\mathcal{M}, s_1)(\sigma_2(t_x^{x'})) = I(\mathcal{M}, s_1)(t_x^{x'}) = I(\mathcal{M}, s)(\sigma_1(t_x^{x'})) = I(\mathcal{M}, s)(t)$
D.h. im Zustand s_2 wird x gleich interpretiert wie $t_x^{x'}$ und es gilt $s_2 \models x = t_x^{x'}$.
Weiterhin gilt:
$s_2(x') = I(\mathcal{M}, s_1)(\sigma_2(x')) = I(\mathcal{M}, s_1)(x') = I(\mathcal{M}, s)(\sigma_1(x')) = I(\mathcal{M}, s)(x)$
und für $y \neq x, y \neq x'$:
$s_2(y) = I(\mathcal{M}, s_1)(\sigma_2(y)) = I(\mathcal{M}, s_1)(y) = I(\mathcal{M}, s)(\sigma_1(y)) = I(\mathcal{M}, s)(y)$.

Es muß nun $s_2 \not\models \varphi$ gelten, da nach Definition der Zuweisung und Definition der Modalität $\subset \cdot \supset$ sonst $s \models \subset x := t \supset \varphi$ gelten würde, was ein Widerspruch zur Annahme $s \not\models \subset x := t \supset \varphi$ darstellt.

Jetzt muß noch gezeigt werden, daß $s_2 \models \Gamma_x^{x'}$ und $s_2 \not\models \Delta_x^{x'}$, denn dann wird die Sequenz $\Gamma_x^{x'},\ \ x = t_x^{x'}\ \ \vdash\ \ \varphi,\ \ \Delta_{\mathrm{x}}^{\mathrm{x}'}$ zu *false* ausgewertet, was wiederum ein Widerspruch zur Annahme der Allgemeingültigkeit der Prämissen ist.

Um $s_2 \models \Gamma_x^{x'}$ und $s_2 \not\models \Delta_x^{x'}$ zu zeigen, wird zuerst Lemma 42 mit der Substitution σ_1 und $\hat{s}(y) = s_1(\sigma_1^{-1}(y)) = s(\sigma_1(\sigma_1^{-1}(y))) = s(y)$ angewandt. Daraus folgt, daß

$s_1 \models \Gamma_x^{x'}$ und $s_1 \not\models \Delta_x^{x'}$, weil

$$s_1 \models \Gamma_x^{x'} \text{ gdw. } s \models \sigma_1(\Gamma_x^{x'}) \text{ gdw. } s \models \Gamma.$$

Da $x \notin var(\Gamma_x^{x'})$ und $x \notin var(\Delta_x^{x'})$ ist Lemma 41 anwendbar und es gilt $s_2 \models \Gamma_x^{x'}$ und $s_2 \not\models \Delta_x^{x'}$.

■

7.3.2 Beweis der Korrektheit der Regel (; rechts) (R68)

$$\frac{\Gamma \ \vdash \ \ll\alpha\gg\varphi,\ \langle\alpha\rangle\ll\beta\gg\varphi,\ \Delta}{\Gamma \ \vdash \ \ll\alpha;\beta\gg\varphi,\ \Delta}$$

Lemma 45.
Die Regel (; rechts) (R68) ist korrekt.

Beweis:
Um die Korrektheit der Regel zu beweisen, ist zu zeigen:
Für alle $\mathcal{M}, s$ gilt:

$$s \models \ll\alpha\gg\varphi \text{ impliziert } s \models \ll\alpha;\beta\gg\varphi$$
$$\text{und}$$
$$s \models \langle\alpha\rangle\ll\beta\gg\varphi \text{ impliziert } s \models \ll\alpha;\beta\gg\varphi.$$

Die erste Implikation ist offensichtlich, interessant ist die zweite.

Nach Definition der Funktion τ ist

$$\begin{aligned}\tau(\langle\alpha\rangle\ll\beta\gg\varphi) = \{s_0 \in W \mid\ & \text{es existiert ein } s \in \phi(\alpha) \text{ mit } s = (s_0, s_1, \ldots, s_m),\\ & s_m \neq \Lambda \text{ und } s_m \in \tau(\ll\beta\gg\varphi)\},\ \text{wobei}\\ \tau(\ll\beta\gg\varphi) = \{t_0 \in W \mid\ & \text{es gibt ein } t \in \phi(\beta) \text{ mit } t = (t_0, t_1, \ldots, t_n) \text{ so daß gilt:}\\ & \text{es gibt ein } t_i \neq \Lambda \text{ mit } t_i \in \tau(\varphi)\}.\end{aligned}$$

Sei nun $s \in \tau(\langle\alpha\rangle\ll\beta\gg\varphi)$. Zu zeigen ist: $s \in \tau(\ll\alpha;\beta\gg\varphi)$.

Nach Definition der Funktion τ ist

$$\begin{aligned}\tau(\ll\alpha;\beta\gg\varphi) = \{s_0 \in W \mid\ & \text{es gint } s \in \phi(\alpha), t \in \phi(\beta) \text{ mit } s_m = t_0 \text{ und}\\ & \text{es gibt } s_i \neq \Lambda \text{ mit } s_i \in \tau(\varphi) \text{ oder}\\ & \text{es gibt } t_i \neq \Lambda \text{ mit } t_i \in \tau(\varphi)\}.\end{aligned}$$

Da $s \in \tau(\langle\alpha\rangle\ll\beta\gg\varphi)$, gibt es ein „passendes“ $t \in \phi(\beta)$ mit einem $t_i \neq \Lambda$ und $t_i \in \tau(\varphi)$. Durch $s_m \in \tau(\ll\beta\gg\varphi)$ ist auch sichergestellt, daß $s_m = t_0$ gilt. Somit ist bewiesen, daß $s \in \tau(\ll\alpha;\beta\gg\varphi)$ gilt.

■

7.4 Relative Vollständigkeit

In diesem Abschnitt soll die relative Vollstädigkeit des Kalküls mit den Modalitäten $\langle\cdot\rangle$, $[\cdot]$, $\subset\cdot\supset$ und $\ll\cdot\gg$ gezeigt werden.
Daß die Modalität $\|\cdot\|$ weggelassen wird, liegt daran, daß es nicht gelungen ist, entsprechende Äquivalenzen oder Ableitungsregeln zu finden, mit denen man die Vollständigkeit beweisen kann. Insbesondere treten die Probleme bei der while-Schleife auf, da man hier keine Invarianz- oder Konvergenzregeln aufstellen kann. Vermutlich ist der Kalkül mit dieser Modalität nicht vollständig.
In [Hare79] ist ein sehr allgemeines Theorem für die Vollständigkeit von Axiomensystemen angegeben, das sich auf verschiedene Logiken anwenden läßt. Hier wird dieses Theorem zum Beweis der relativen Vollständigkeit der Prozeßlogik erster Stufe verwendet, allerdings nicht in der allgemeinen Version, sondern angepaßt an die Prozeßlogik.

Theorem 46 (Vollständigkeitstheorem).
Ein Kalkül für die Prozeßlogik erster Stufe ist vollständig für arithmetische Strukturen[16]*, wenn folgende Bedigungen erfüllt sind:*

1. *Der Kalkül ist aussagenlogisch vollständig, d.h., alle Instanzen aussagenlogischer Tautologien sind Axiome und Modus Ponens ist als Ableitungsregel vorhanden.*
2. *Jede Formel aus Fml_{pl} ist prädikatenlogisch ausdrückbar, d.h. für jede Formel $\varphi \in Fml_{pl}$ gibt es eine Formel $G \in PL$ über jeder arithmetischen Signatur, so daß $\models \varphi \leftrightarrow G$ gilt.*
3. *Für alle Variablen x, Programme α, Formeln $\varphi, \psi \in Fml_{pl}$ und alle Modalitäten $M \in \{\langle\cdot\rangle, [\cdot], \subset\cdot\supset, \ll\cdot\gg\}$*
   (a) *wenn $\vdash \varphi \to \psi$ dann $\vdash M(\alpha)\varphi \to M(\alpha)\psi$,*
   (b) *wenn $\vdash \varphi \to \psi$ dann $\vdash \exists x\varphi \to \exists x\psi$,*

[16] Vollständig für arithmetische Strukturen bedeutet, daß für alle Formeln $\varphi \in Fml_{pl}$ gilt: wenn $\models_{\mathcal{M}} \varphi$ für alle arithmetischen Strukturen $\mathcal{M}$, dann $\vdash \varphi$.

4. *Für alle Programme α, Formeln $F, G \in PL$ und alle Modalitäten $M \in \{\langle\cdot\rangle, [\cdot], \subset\cdot\supset, \ll\cdot\gg\}$*
   (a) *wenn $\models F$ dann $\vdash F$,*
   (b) *wenn $\models F \to M(\alpha)G$ dann $\vdash F \to M(\alpha)G$,*
   (c) *wenn $\models F \to \neg M(\alpha)G$ dann $\vdash F \to \neg M(\alpha)G$.*

Beweis: Siehe [Hare79]

Das folgende Theorem 47 garantiert, daß die Voraussetzung 2 des Vollständigkeitstheorems erfüllt ist.

Theorem 47.
Sei $F \in Fml_{pl}$. Dann gibt es eine Formel $F_{\mathcal{A}}$ der Prädikatenlogik erster Stufe über jeder arithmetischen Signatur mit $\models F \leftrightarrow F_{\mathcal{A}}$.

Dieses Theorem sagt aus, daß über den arithmetischen Strukturen die Prozeßlogik erster Stufe durch die Prädikatenlogik erster Stufe ausgedrückt werden kann, oder anders gesagt: über arithmetischen Strukturen ist die Prozeßlogik erster Stufe nicht ausdrucksstärker als die Prädikatenlogik erster Stufe.

Beweis:
Was für den Beweis von entscheidender Bedeutung sein wird, ist die Möglichkeit, in der Arithmetik endliche Folgen von Elementen des Universums in einem einzigen Element zu kodieren. Man kann also ein Prädikat $Encode(x, i, y)$ definieren. Das heißt, es gibt eine prädikatenlogische Formel $\mathcal{G}$, so daß für jede arithmetische Struktur $\mathcal{A}$ und jeden Zustand s gilt:

$$\begin{array}{lll}(\mathcal{A}, s) \models \mathcal{G} \to Encode(x, i, y) \textit{ gilt} & \textit{gdw.} & I(\mathcal{A}, s)(i) \in \mathbb{N} \textit{ und} \\ & & I(\mathcal{A}, s)(y) \in \mathbb{N} \textit{ und} \\ & & gödel^{-1}(I(\mathcal{A}, s)(y)) = (x_1, \ldots, x_n) \textit{ und} \\ & & I(\mathcal{A}, s)(i) \leq n \textit{ und} \\ & & I(\mathcal{A}, s)(x) = x_{I(\mathcal{A},s)(i)}.\end{array}$$

Dabei ist *gödel* eine injektive Abbildung der Menge der endlichen Folgen über $\mathcal{U}_{\mathcal{A}}$ in $\mathbb{N}$.
Die Intuition dabei ist, daß $Encode(x, i, y)$ genau dann gilt, wenn x die i-te Komponente von y ist.
Der besseren Lesbarkeit wegen wird außerdem ein zweistelliges Funktionssymbol *nth* definiert, für das gelten soll:

$$\models_{\mathcal{A}} \mathcal{G} \to (nat(z) \wedge nat(n) \to (nth(z, n) = z_n \leftrightarrow Encode(z_n, n, z) = true)).$$

$nth(z, n)$ liefert also die n-te Komponente der in z kodierten Folge.
Ein weiteres einstelliges Funktionssymbol, das verwendet wird und das durch $\mathcal{G}$ definiert sei, ist *length*, welches die Länge der im Argument kodierten Folge liefert. Eine Folge der Länge Null, d.h. $length(z) = 0$, bedeutet, daß die Folge z nur aus einem Element (Zustand) besteht.

Der Beweis wird durch strukturelle Induktion über den Aufbau der zu kodierenden Formel F geführt.
Seien $F, G, H \in Fml_{pl}$. Zu zeigen ist, daß eine prädikatenlogische Formel $F_{\mathcal{A}}$ existiert mit $\models_{\mathcal{A}} F \leftrightarrow F_{\mathcal{A}}$.

Induktionsanfang:

- $F = true$, $F = false$, F atomar:
  trivial, da F bereits prädikatenlogisch ist.

Induktionsschritt:

- $F = \neg G$:
  Nach Induktionsannahme existiert $G_{\mathcal{A}}$ mit $\models_{\mathcal{A}} G \leftrightarrow G_{\mathcal{A}}$. Damit gilt $\models_{\mathcal{A}} F \leftrightarrow F_{\mathcal{A}}$ mit $F_{\mathcal{A}} = \neg G_{\mathcal{A}}$.

- $F = G \vee H$ oder $F = \exists x G$:
  analog zu $F = \neg G$.

- $F = \langle\alpha\rangle G$:
  Nach Induktionsvoraussetzung existiert $G_{\mathcal{A}}$ mit $\models_{\mathcal{A}} G \leftrightarrow G_{\mathcal{A}}$. Sei z der (endliche) Vektor der Variablen, die in α enthalten sind, und sei z' ein Vektor gleicher Länge von Variablen, die nicht in α enthalten sind. Im folgenden gelte die Konvention, daß die Variable x aus z der Variable x' aus z' entspricht. Durch strukturelle Induktion über den Programmaufbau von α wird nun bewiesen, daß es eine prädikatenlogische Formel $F_{\mathcal{A}}$ gibt mit $\models_{\mathcal{A}} \langle\alpha\rangle G \leftrightarrow F_{\mathcal{A}}$. Dazu wird eine prädikatenlogische Formel $F_\alpha(z, z')$ konstruiert, für die gilt:

  $s_0 \models F_\alpha(z, z')$ *gdw.*
  es gibt $(s_0, s_1, \ldots) \in \phi(\alpha)$ *und es gibt ein* $s_i \in (s_0, s_1, \ldots)$ *mit* $s_i(z) = z'$.

  Diese Formel drückt aus, daß der Zustand z' durch das Programm α erreichbar ist, wenn α im Zustand z gestartet wird, wobei nicht gesagt ist, daß α terminieren muß.

  Ferner wird noch eine Formel $\hat{F}_\alpha(z, z')$ konstruiert, für die gilt:

  $$s_0 \models \hat{F}_\alpha(z, z') \textit{ gdw. es gibt } (s_0, s_1, \ldots, s_n) \in \phi(\alpha) \textit{ mit } s_n(z) = z'.$$

Diese Formel drückt aus, daß das Programm α, im Zustand z gestartet, im Zustand z' terminiert.

Da bei der Modalität $\langle\cdot\rangle$ nur der Endzustand der Programmausführung von Interesse ist, gilt:

$$\models_{\mathcal{A}} \langle\alpha\rangle G \leftrightarrow \exists z'(\hat{F}_\alpha(z,z') \wedge (G_{\mathcal{A}})_z^{z'})$$

Es bleibt zu zeigen, daß $\hat{F}_\alpha$ für alle α existiert, was durch folgende induktive Definition geschieht:

- $\alpha = (x := t)$:
  Setze $\hat{F}_{x:=t}(x,x') := x' = t$.
- $\alpha = (P?)$:
  Setze $\hat{F}_{P?}(z,z') := P$.

- $\alpha = (\beta;\gamma)$:
  Setze $\hat{F}_{\beta;\gamma}(z,z') := \exists z''(\hat{F}_\beta(z,z'') \wedge \hat{F}_\gamma(z'',z'))$, *wobei* z'' *neu.*
- $\alpha = (\beta \cup \gamma)$:
  Setze $\hat{F}_{\beta\cup\gamma}(z,z') := \hat{F}_\beta(z,z') \vee \hat{F}_\gamma(z,z')$.
- $\alpha = (\beta^*)$:
  Setze $\hat{F}_{\beta^*}(z,z') := \mathcal{G} \rightarrow (\exists z^*(nth(z^*,0) = z \wedge nth(z^*,length(z^*)) = z' \wedge$
  $\forall n(nat(n) \wedge n < length(z^*) \rightarrow$
  $\hat{F}_\beta(nth(z^*,n), nth(z^*,n+1)))))$.

- $F = [\alpha]G$:
  Nach Definiton gilt $[\alpha]G = \neg\langle\alpha\rangle\neg G$. Damit trifft einer der anderen Fälle zu.

- $F = \subset\alpha\supset G$:
  Es wird dieselbe Formel $\hat{F}_\alpha$ wie oben verwendet. Die Formel F_α wird zudem wie folgt definiert:

  - $\alpha = (x := t)$:
    Setze $F_{x:=t}(x,x') := x' = t$.
  - $\alpha = (P?)$:
    Setze $F_{P?}(z,z') := P$.

  - $\alpha = (\beta;\gamma)$:
    Setze $F_{\beta;\gamma}(z,z') := \mathcal{G} \rightarrow (F_\beta(z,z') \vee \exists z''(\hat{F}_\beta(z,z'') \wedge F_\gamma(z'',z')))$, *wobei* z'' *neu.*
  - $\alpha = (\beta \cup \gamma)$:
    Setze $F_{\beta\cup\gamma}(z,z') := F_\beta(z,z') \vee F_\gamma(z,z')$.

– $\alpha = (\beta^*)$:
Setze $F_{\beta^*}(z,z') := \mathcal{G} \to (\exists z''(\hat{F}_{\beta^*}(z,z'') \wedge F_\beta(z'',z')))$.

Dann gilt:

$$\models_{\mathcal{A}} \subset\alpha\supset G \leftrightarrow (\mathcal{G} \to \forall z'(F_\alpha(z,z') \to (G_{\mathcal{A}})^z_z))$$

Die Gültigkeit der angegebenen Äquivalenzen folgt direkt aus der Definition der Semantik.

■

Bemerkung:
Wie schon erwähnt ist es nicht gelungen, die Vollständigkeit des Kalküls mit der Modalität $\|\cdot\|$ zu beweisen. Aus diesem Grunde wurde auch keine Kodierung in Prädikatenlogik angegeben. Dazu wäre eine andere Definition der Formel F_α erforderlich, die Zustandsfolgen betrachtet und nicht nur ein Paar von Zuständen.

Ferner ist zu beachten, daß die Transformation von prozeßlogischen Formeln in prädikatenlogische Formeln für die nichtdeterministischen Verknüpfungszeichen $\cup$ und * angegeben wurde und nicht direkt für Formeln, deren Programme in der Programmiersprache **P** notiert sind. Da aber die Konstrukte von **P** nur Abkürzungen der im Beweis verwendeten Verknüpfungszeichen darstellen, lassen sich selbstverständlich auch Formeln mit Programmen in **P** durch prädikatenlogische Formeln ausdrücken.

Die Voraussetzung 1 (aussagenlogische Vollständigkeit) des Theorems 46 wird durch die Regeln (Einführung von Tautologien als Axiome) (R1) und (Schnitt) (R7) gewährleistet. Aus letzterer ist der Modus Ponens ableitbar.

Dafür, daß die Voraussetzung 3 der Vollständigkeitstheorems erfüllt ist, sorgen die Regeln (Generalisierung) (R33), (R48), (R65), (R80) und (Generalisierung') (R14).

Schließlich muß noch gezeigt werden, daß die Voraussetzungen 4 des Theorems 46 erfüllt sind. Teil (a) ist klar, da alle gültigen Formeln der Prädikatenlogik erster Stufe als Axiome zur Verfügung stehen (Regel (R2)). Daß die Teile (b) und (c) erfüllt sind, garantieren die Theoreme 48–51. Eigentlich wären für vier Modalitäten acht Theoreme aufzustellen, aber dadurch, daß die Modalitäten $\langle\cdot\rangle$ und $[\cdot]$ bzw. $\subset\cdot\supset$ und $\ll\cdot\gg$ komplementär zueinander sind, genügen die vier folgenden Theoreme.

Theorem 48 ($\langle\cdot\rangle$-Vollständigkeit).
Für jedes Programm α und alle Formeln F, G der Prädikatenlogik erster Stufe gilt:

$$\textit{wenn} \models F \to \langle\alpha\rangle G \;\textit{dann} \vdash F \to \langle\alpha\rangle G.$$

Beweis: Der Beweis wird durch Induktion über den Aufbau des Programms α geführt. Seien α, β, γ Programme der Programmiersprache **P** und F, G Formeln der Prädikatenlogik erster Stufe über einer arithmetischen Signatur.

Induktionsanfang:

- $\alpha = (x := t)$
  Gilt $\models F \to \langle x := t\rangle G$, so gilt nach Definition der Semantik der Prozeßlogik auch $\models F \to G_x^t$[17], was rein prädikatenlogisch und mit Regel (R2) ableitbar ist. Somit ist mit Regel (R16) auch $\vdash \; F \to \langle x := t\rangle G$ ableitbar[18].

Induktionsschluß:

- $\alpha = (\mathtt{IF}\ \epsilon\ \mathtt{THEN}\ \beta\ \mathtt{ELSE}\ \gamma)$
  Gilt $\models F \to \langle\mathtt{IF}\ \epsilon\ \mathtt{THEN}\ \beta\ \mathtt{ELSE}\ \gamma\rangle G$, so gilt nach Definition der Semantik der Prozeßlogik auch $\models F \wedge \epsilon \to \langle\beta\rangle G$ und $\models F \wedge \neg\epsilon \to \langle\gamma\rangle G$. Damit hat man nach Induktionsannahme $F,\ \epsilon \ \vdash \ \langle\beta\rangle G$[19] und $F,\ \neg\epsilon \ \vdash \ \langle\gamma\rangle G$, woraus mit Regel (if-then-else rechts) (R20)

  $$F \ \vdash \ \langle\mathtt{IF}\ \epsilon\ \mathtt{THEN}\ \beta\ \mathtt{ELSE}\ \gamma\rangle G$$

  ableitbar ist. Daraus läßt sich leicht

  $$\vdash \ F \to \langle\mathtt{IF}\ \epsilon\ \mathtt{THEN}\ \beta\ \mathtt{ELSE}\ \gamma\rangle G$$

  ableiten.

- $\alpha = (\beta; \gamma)$
  Gilt $\models F \to \langle\beta;\gamma\rangle G$, so gilt nach Definition der Prozeßlogik $\models F \to \langle\beta\rangle\langle\gamma\rangle G$. Da prozeßlogische Formeln in der Prädikatenlogik ausdrückbar sind, gibt

[17] Durch eine gebundene Umbenennung ist sicherzustellten, daß x nur frei in G vorkommt.

[18] Die Ableitung ist nicht direkt in einem Schritt mit der Regel (R16) durchführbar und soll hier auch nicht vorgeführt werden.

[19] Man erhält diese Sequenz nicht direkt durch einen Ableitungsschritt, sondern man erhält $\vdash \ F \wedge \epsilon \to \langle\alpha\rangle G$, woraus man aber offensichtlich die besagte Sequenz ableiten kann. Im folgenden wird dieser Ableitungsschritt immer implizit vorausgesetzt.

es eine prädikatenlogische Formel H, die äquivalent ist zu $\langle\gamma\rangle G$. Also gilt $\models F \to \langle\beta\rangle H$ und damit ist nach Induktionsannahme

$$F \;\vdash\; \langle\beta\rangle H \tag{1}$$

ableitbar. Offensichtlich gilt $\models H \to \langle\gamma\rangle G$ und nach Induktionsannahme ist

$$H \;\vdash\; \langle\gamma\rangle G$$

ableitbar. Durch Anwendung der Regel (Generalisierung) (R33) kann man daraus

$$\langle\beta\rangle H \;\vdash\; \langle\beta\rangle\langle\gamma\rangle G \tag{2}$$

ableiten.

Schließlich erhält man aus (1) und (2)

$$\vdash\; F \to \langle\beta\rangle\langle\gamma\rangle G$$

und daraus mit Regel (; rechts) (R18)

$$\vdash\; F \to \langle\beta;\gamma\rangle G.$$

- $\alpha = (\texttt{WHILE}\ \epsilon\ \texttt{DO}\ \beta)$
  Um die Ableitbarkeit zu zeigen, wird zuerst eine rein prädikatenlogische Formel $L_{\epsilon,\beta,G}(n)$ definiert, die ausdrückt, daß die Schleife $\texttt{WHILE}\ \epsilon\ \texttt{DO}\ \beta$ nach genau n Schritten terminiert und daß im Endzustand G gilt.

  Die Formel $L_{\epsilon,\beta,G}(n)$ wird definiert durch

$$\begin{aligned} L_{\epsilon,\beta,G}(n) := \mathcal{G} \to\ & (\exists z^*(length(z^*) = n \land nth(z^*,0) = z \land \forall m((nat(m)\land \\ & m < n \land z_m = nth(z^*,m) \land z_{m+1} = nth(z^*,m+1)) \to \\ & (\hat{F}_\beta(z_m, z_{m+1}) \land \epsilon_z^{z_m})) \\ & \land(z_n = nth(z^*,n) \to (\neg\epsilon_z^{z_n}) \land G_z^{z_n}))). \end{aligned}$$

  und es gilt

$$\models \forall n(L_{\epsilon,\beta,G}(n) \to \langle\texttt{WHILE}\ \epsilon\ \texttt{DO}\ \beta\rangle G).$$

  Nun wird durch Induktion auf der Objektebene (R15)

$$\vdash\; \forall n(L_{\epsilon,\beta,G}(n) \to \langle\texttt{WHILE}\ \epsilon\ \texttt{DO}\ \beta\rangle G) \tag{3}$$

  abgeleitet.

  Es gilt

$$\models L_{\epsilon,\beta,G}(0) \to (G \land \neg\epsilon)$$

und damit $\models L_{\epsilon,\beta,G}(0) \to G$ und $\models L_{\epsilon,\beta,G}(0) \to \neg\epsilon$. Da beide Formeln rein prädikatenlogisch sind, kann man sie ableiten. Nun wendet man die Regel (while exit rechts) (R28) und erhält

$$L_{\epsilon,\beta,G}(0) \;\vdash\; \langle \texttt{WHILE}\ \epsilon\ \texttt{DO}\ \beta\rangle G. \tag{4}$$

Die Sequenz (4) stellt den Induktionsanfang dar.

Den Induktionsschluß leitet man wie folgt ab:

Nach der Definition der Semantik von "while" gilt

$$\models\;\; L_{\epsilon,\beta,G}(n+1) \to \langle\beta\rangle L_{\epsilon,\beta,G}(n)$$

und somit ist nach Induktionsannahme

$$L_{\epsilon,\beta,G}(n+1) \;\vdash\; \langle\beta\rangle L_{\epsilon,\beta,G}(n) \tag{5}$$

ableitbar, da β von geringerer Komplexität als α ist. Die Induktionsannahme der Induktion auf der Objektebene ist

$$L_{\epsilon,\beta,G}(n) \;\;\to\;\; \langle \texttt{WHILE}\ \epsilon\ \texttt{DO}\ \beta\rangle G,$$

woraus man

$$L_{\epsilon,\beta,G}(n) \;\vdash\; \langle \texttt{WHILE}\ \epsilon\ \texttt{DO}\ \beta\rangle G \tag{6}$$

und mit der Regel (Generalisierung) (R33)

$$\langle\beta\rangle L_{\epsilon,\beta,G}(n) \;\vdash\; \langle\beta\rangle\langle \texttt{WHILE}\ \epsilon\ \texttt{DO}\ \beta\rangle G \tag{7}$$

erhält. Aus (5) und (7) erhält man

$$L_{\epsilon,\beta,G}(n+1) \;\vdash\; \langle\beta\rangle\langle \texttt{WHILE}\ \epsilon\ \texttt{DO}\ \beta\rangle G. \tag{8}$$

Weiterhin gilt $\models L_{\epsilon,\beta,G}(n+1) \to \epsilon$, woraus man

$$L_{\epsilon,\beta,G}(n+1) \;\vdash\; \epsilon \tag{9}$$

ableiten kann. Mit der Regel (while rechts (unwind)) (R27) leitet man aus (8) und (9)

$$L_{\epsilon,\beta,G}(n+1) \;\vdash\; \langle \texttt{WHILE}\ \epsilon\ \texttt{DO}\ \beta\rangle G \tag{10}$$

ab. Da es nun gelungen ist, (6) und (10) abzuleiten, kann man

$$L_{\epsilon,\beta,G}(n) \to \langle \texttt{WHILE}\ \epsilon\ \texttt{DO}\ \beta\rangle G \;\vdash\; L_{\epsilon,\beta,G}(n+1) \to \langle \texttt{WHILE}\ \epsilon\ \texttt{DO}\ \beta\rangle G \tag{11}$$

ableiten.[20] Schließlich kann man aus (4) und (11) mit der Regel (Induktion) (R15) die Sequenz (3) ableiten.

Offensichtlich gilt $\models F \to \exists n L_{\epsilon,\beta,G}(n)$. Man kann also

$$F \;\vdash\; \exists n L_{\epsilon,\beta,G}(n) \tag{12}$$

ableiten, da diese Formel rein prädikatenlogisch ist. Schließlich erhält man die gesuchte Sequenz $\vdash F \to \langle \texttt{WHILE}\ \epsilon\ \texttt{DO}\ \beta\rangle G$ aus (3) und (12) durch Anwendung der Regeln für die Quantoren.

■

Theorem 49 ([·]-Vollständigkeit).
Für jedes Programm α und alle Formeln F, G der Prädikatenlogik erster Stufe F, G gilt:

$$\textit{wenn} \models F \to [\alpha]G \;\textit{dann} \vdash F \to [\alpha]G.$$

Beweis:
Der Beweis verläuft analog zu dem Beweis des Theorems 48 und wird ebenfalls durch Induktion über den Programmaufbau von α geführt. Die Fälle $\alpha = (x := t)$, $\alpha = (\texttt{IF}\ \epsilon\ \texttt{THEN}\ \beta\ \texttt{ELSE}\ \gamma)$ und $\alpha = (\beta;\gamma)$ sind völlig analog beweisbar. Die Ableitbarkeit von

$$\vdash\; F \to [\texttt{WHILE}\ \epsilon\ \texttt{DO}\ \beta]G \tag{1}$$

wird durch Anwendung der Regel (while rechts (Invarianz)) (R 26) gezeigt. Die dazu benötigte Invariante $Inv_{\epsilon,\beta,P}$ wird definiert durch

$$Inv_{\epsilon,\beta,G} := \mathcal{G} \to \forall z'(\hat{F}_\beta(z,z') \to G_z^{z'}),$$

was aussagt, daß die Schleife entweder terminiert und dann G gilt oder nicht terminiert. Die Formel $L_{\epsilon,\beta,G}$ ist dabei wie im Beweis von Theorem 48 definiert.

Ableitbar sind nun

$$F \;\vdash\; Inv_{\epsilon,\beta,G} \tag{2}$$

$$Inv_{\epsilon,\beta,G},\; \neg\epsilon \;\vdash\; G, \tag{3}$$

da $F \to Inv_{\epsilon,\beta,G}$ bzw. $Inv_{\epsilon,\beta,G} \wedge \neg\epsilon \to G$ allgemeingültig und rein prädikatenlogisch sind.

[20] Dieser Ableitungsschritt ist nicht ganz trivial. Man muß dazu einen Cut oder eine Lemma-Regel anwenden.

Da β von geringerer Komplexität als α und $Inv_{\epsilon,\beta,G} \wedge \epsilon \to [\beta]Inv_{\epsilon,\beta,G}$ allgemeingültig ist, kann man nach Induktionsannahme auch

$$Inv_{\epsilon,\beta,G},\ \ \epsilon\ \ \vdash\ \ [\beta]Inv_{\epsilon,\beta,G} \tag{4}$$

ableiten.

Die Sequenzen (2), (3) und (4) entsprechen den Prämissen der Regel (while rechts (Invarianz)) (R26), mit der man nun (1) ableiten kann.

■

Theorem 50 ($\ll \cdot \gg$-Vollständigkeit).
Für jedes Programm α und alle Formeln F, G der Prädikatenlogik erster Stufe gilt:

$$\textit{wenn} \models F \to \ll\alpha\gg G \ \textit{dann} \vdash F \to \ll\alpha\gg G.$$

Beweis: Der Beweis wird durch Induktion über den Aufbau des Programms α geführt. Seien α, β, γ Programme der Programmiersprache **P** und F, G Formeln der Prädikatenlogik erster Stufe über einer arithmetischen Signatur.

Induktionsanfang:

- $\alpha = (x := t)$
  Gilt $\models F \to \ll x := t\gg G$, so gilt nach Definition der Semantik der Prozeßlogik auch $\models F \to G$ oder $\models F \to G^t_x$, was beides rein prädikatenlogisch und mit Regel (R 2) ableitbar ist. Somit ist mit Regel (R66) auch $\vdash\ \ F \to \ll x := t\gg G$ ableitbar.

Induktionsschluß:

- $\alpha = (\texttt{IF}\ \epsilon\ \texttt{THEN}\ \beta\ \texttt{ELSE}\ \gamma)$
  Analog zum Beweis des Theorems 48.

- $\alpha = (\beta; \gamma)$
  Gilt $\models F \to \ll\beta;\gamma\gg G$, so gilt nach Definition der Prozeßlogik $\models F \to \ll\beta\gg G$ oder $\models F \to \langle\beta\rangle\ll\gamma\gg G$. Damit kann man

  $$\vdash\ \ F \to \ll\beta\gg G \tag{1}$$

  oder

  $$\vdash\ \ F \to \langle\beta\rangle\ll\gamma\gg G \tag{2}$$

  ableiten. Sequenz (1) kann man nach Induktionsannahme ableiten und (2) analog zu der Sequenz $\vdash F \to \langle\beta\rangle\langle\gamma\rangle G$ im Beweis des Theorems 48.

- $\alpha = (\texttt{WHILE}\ \epsilon\ \texttt{DO}\ \beta)$
  Ähnlich wie vorher[21] wird eine Formel $K_{\epsilon,\beta,G}(n)$ definiert durch
  $$\begin{aligned} K_{\epsilon,\beta,G}(n) := \mathcal{G} \to \quad & (\exists z^*(length(z^*) = n \wedge nth(z^*, 0) = z \wedge \\ & \forall m((nat(m) \wedge m < n \wedge z_m = nth(z^*, m) \wedge \\ & z_{m+1} = nth(z^*, m+1)) \to (\hat{F}_\beta(z_m, z_{m+1}) \wedge \epsilon_z^{z_m}) \wedge \\ & (z_n = nth(z^*, n) \to ((G_z^{z_n} \vee \epsilon_z^{z_n}) \wedge \\ & \exists z'(F_\beta(nth(z^*, n), z') \wedge G_z^{z'})))))). \end{aligned}$$
  Die Formel $K_{\epsilon,\beta,G}(n)$ drückt aus, daß die Schleife mindestens $n \geq 0$ mal läuft und dann der Zustand z_n eingenommen wird. In diesem Zustand z_n gilt nun G oder es gilt ϵ. Im letzteren Fall läuft die Schleife noch mindestens ein weiteres Mal und während der $(n+1)$. Ausführung des Schleifenrumpfes β wird ein Zustand z' eingenommen, in dem G gilt.

  Der Beweis läuft im Prinzip gleich ab wie vorher, d.h. durch Anwendung der Induktionsregel (R15) wird $\vdash \forall n(K_{\epsilon,\beta,G}(n) \to \ll\texttt{WHILE}\ \epsilon\ \texttt{DO}\ \beta\gg G)$ abgeleitet.

  Der Induktionsanfang für die Objektebene ist
  $$K_{\epsilon,\beta,G}(0) \ \vdash \ \ll\texttt{WHILE}\ \epsilon\ \texttt{DO}\ \beta\gg G. \tag{3}$$
  Es gilt
  $$\models K_{\epsilon,\beta,G}(0) \to (G \vee \epsilon) \wedge \ll\beta\gg G,$$
  also
  $$\models K_{\epsilon,\beta,G}(0) \wedge \neg\epsilon \to G$$
  und
  $$\models K_{\epsilon,\beta,G}(0) \to \ll\beta\gg G.$$
  Also sind die Voraussetzungen von Regel (while rechts) (R76) ableitbar und damit (3).

  Die Induktionsannahme ist
  $$K_{\epsilon,\beta,G}(n) \to \ll\texttt{WHILE}\ \epsilon\ \texttt{DO}\ \beta\gg G, \tag{4}$$
  woraus man durch Anwendung der Regel (Generalisierung) (R33) sofort
  $$\langle\beta\rangle K_{\epsilon,\beta,G}(n) \ \vdash \ \langle\beta\rangle\ll\texttt{WHILE}\ \epsilon\ \texttt{DO}\ \beta\gg G \tag{5}$$
  erhält.

[21] Mit „vorher“ sei im folgenden der Beweis des Theorems 48 gemeint.

Aus der Definition der Semantik der Prozeßlogik und der Formel $K_{\epsilon,\beta,G}$ folgt

$$\models K_{\epsilon,\beta,G}(n+1) \to \langle\beta\rangle K_{\epsilon,\beta,G}(n). \tag{6}$$

Zudem gilt $\models K_{\epsilon,\beta,G}(n+1)\epsilon$.

Nach Induktionsannahme sind die beiden Sequenzen

$$K_{\epsilon,\beta,G}(n+1) \;\vdash\; \langle\beta\rangle K_{\epsilon,\beta,G}(n) \tag{7}$$

$$K_{\epsilon,\beta,G}(n+1) \;\vdash\; \epsilon. \tag{8}$$

ableitbar.

Aus (7) erhält man mit (5)

$$K_{\epsilon,\beta,G}(n+1) \;\vdash\; \langle\beta\rangle\ll\texttt{WHILE}\ \epsilon\ \texttt{DO}\ \beta\gg G \tag{9}$$

und aus (8) und (9) mit der Regel (while rechts (unwind)) (R78)

$$K_{\epsilon,\beta,G}(n+1) \;\vdash\; \ll\texttt{WHILE}\ \epsilon\ \texttt{DO}\ \beta\gg G. \tag{10}$$

Insgesamt hat man (10) aus (4) abgeleitet. Damit und mit der Ableitbarkeit von (3) hat man die Voraussetzungen für die Anwendung der Induktionsregel (R15). Diese ergibt

$$\vdash\; \forall n(K_{\epsilon,\beta,G}(n) \to \ll\texttt{WHILE}\ \epsilon\ \texttt{DO}\ \beta\gg G). \tag{11}$$

Zudem gilt wegen $\models F \to \ll\texttt{WHILE}\ \epsilon\ \texttt{DO}\ \beta\gg G$

$$\models F \to \exists n K_{\epsilon,\beta,G}(n)$$

und also ist

$$\vdash\; F \to \exists n K_{\epsilon,\beta,G}(n) \tag{12}$$

ableitbar. Aus (11) und (12) erhält man mit den Quantorenregeln

$$\vdash\; F \to \ll\texttt{WHILE}\ \epsilon\ \texttt{DO}\ \beta\gg G.$$

∎

Theorem 51 ($\subset \cdot \supset$-Vollständigkeit).
Für jedes Programm α und alle Formeln F, G der Prädikatenlogik erster Stufe gilt:

$$\textit{wenn} \models F \to \subset\alpha\supset G \;\; \textit{dann} \vdash F \to \subset\alpha\supset G.$$

Beweis:

Der Beweis verläuft analog zu dem Beweis des Theorems 50 und wird ebenfalls durch Induktion über den Programmaufbau von α geführt. Die Fälle $\alpha = (x := t)$, $\alpha = (\texttt{IF}\ \epsilon\ \texttt{THEN}\ \beta\ \texttt{ELSE}\ \gamma)$ und $\alpha = (\beta; \gamma)$ sind völlig analog. Bleibt noch der Fall $\alpha = (\texttt{WHILE}\ \epsilon\ \texttt{DO}\ \beta)$ zu betrachten:

Dazu wird analog zum Beweis des Theorems 49 eine Formel $Inv_{\epsilon,\beta,P}$ definiert durch

$$Inv_{\epsilon,\beta,G} := \mathcal{G} \to \forall z'(F_\beta(z, z') \to G_z^{z'})$$

Die Invariante $Inv_{\epsilon,\beta,G}$ drückt aus, daß während eines Schleifendurchlaufes G in jedem Zustand gilt.

Analog zum Beweis des Theorems 49 lassen sich folgende vier Sequenzen ableiten, da die entsprechenden Formeln allgemeingültig sind:

$$\begin{array}{rcl} F & \vdash & Inv_{\epsilon,\beta,G} \\ Inv_{\epsilon,\beta,G},\ \neg\epsilon & \vdash & G \\ Inv_{\epsilon,\beta,G},\ \epsilon & \vdash & [\beta] Inv_{\epsilon,\beta,G} \\ Inv_{\epsilon,\beta,G},\ \epsilon & \vdash & \subset\beta\supset G. \end{array}$$

Diese vier Sequenzen stellen die Prämissen der Regel (while rechts (Invarianz)) (R44) dar. Damit ist die gesuchte Sequenz

$$\vdash\ F \to \subset\texttt{WHILE}\ \epsilon\ \texttt{DO}\ \beta\supset G$$

ableitbar.

■

8 Beispiele

Zum Abschluß dieser Studienarbeit soll noch an einem Beispiel gezeigt werden, wozu diese Erweiterung der Dynamischen Logik verwendet werden kann. Die erste Anwendungsmöglichkeit, die man sich vielleicht vorstellt, ist die Verwendung der Modalität $\subset \cdot \supset$, um zu garantieren, daß eine Schleifeninvariante tatsächlich während der ganzen Schleife gilt. Das ist aber in der Praxis für gewöhnlich nicht der Fall, da Schleifeninvarianten meistens nur am Ende und am Anfang der Schleife gelten. Also genügt für diesen Fall die Modalität $[\cdot]$, die ja bereits die herkömmliche Dynamische Logik bietet. Es kann aber durchaus passieren, daß die Schleifeninvariante sehr kompliziert ist oder man sie überhaupt nicht kennt. Ist dies der Fall, dann kann man nicht mehr beweisen, daß die Schleife tatsächlich das tut, was man sich vorstellt. Um aber trotzdem ein gewisses Maß an Sicherheit zu erlangen, kann man sich vorstellen, daß man eine einfache Invariante findet, die zwar nicht ausreicht, um die Korrektheit zu beweisen, aber dennoch gewisse Anforderungen garantiert, zum Beispiel daß eine Variable während der Ausführung niemals kleiner Null wird oder daß Array-Grenzen eingehalten werden. Es ist oft sehr einfach, solche schwachen Invarianten zu finden, die dann wirklich während der ganzen Ausführung der Schleife gelten müssen und nicht nur am Anfang und Ende, was sie in gewisser Weiser auch wieder stärker macht. Die Gültigkeit solcher schwachen Invarianten während der gesamten Ausführung kann mit Hilfe der Modalität $\subset \cdot \supset$ bewiesen werden.
Dieser eben geschilderte Fall soll nun an einem einfachen Beispiel demonstriert werden. Gegeben sei folgendes Programm:

```
s:=0
IF even(x) THEN
   WHILE x ≠ 0 DO
      s:=s+x
      x:=x-2
```

Dieses Programm soll für alle Eingaben `x`, die gerade und größer Null sind, die Summe der geraden Zahlen von Null bis `x` berechnen. Um dies zu beweisen, würde man in der klassischen Dynamischen Logik zuerst versuchen, eine ausreichend starke Schleifeninvariante zu finden. Hier bietet es sich offensichtlich an, eine geschlossene Formel zu suchen, die das Gewünschte berechnet. Für dieses kleine Programm ist das noch relativ einfach. Eine geschlossene Formel für die Summe der geraden positiven Zahlen von Null bis n ist $\frac{n(n+2)}{4}$.
Zu zeigen ist jetzt die Gültigkeit der Sequenz

$$even(x),\ \ x \geq 0,\ \ y = x\ \ \vdash$$
$$[\texttt{s:=0}][\texttt{IF even(x) THEN WHILE x} \neq \texttt{0 DO s:=s+x; x:=x-2}]s = \frac{y(y+2)}{4}$$

In einem ersten Ableitungsschritt erhält man durch Anwendung der Regel (Zuweisung rechts) (R16) die neue Beweisverpflichtung

$$even(x),\ \ x \geq 0,\ \ s = 0,\ \ y = x\ \ \vdash$$
$$[\texttt{IF even(x) THEN WHILE x} \neq \texttt{0 DO s:=s+x; x:=x-2}]s = \tfrac{y(y+2)}{4}.$$

Jetzt ist die Regel (if positive rechts) (R21) anzuwenden und man bekommt die zwei Beweisverpflichtungen

$$even(x),\ \ x \geq 0,\ \ s = 0,\ \ y = x\ \ \vdash\ \ even(x)$$

und

$$even(x),\ \ x \geq 0,\ \ s = 0,\ \ y = x\ \ \vdash$$
$$[\texttt{WHILE x} \neq \texttt{0 DO s:=s+x; x:=x-2}]s = \tfrac{y(y+2)}{4},$$

wobei die erste trivialerweise erfüllt ist.
Im nächsten Schritt gilt es, die while-Schleife zu beseitigen. Dazu wird die Regel (while rechts (Invarianz)) (R26) verwendet. Als Invariante wird $I = even(x) \wedge x \geq 0 \wedge \frac{y(y+2)}{4} = s + \frac{x(x+2)}{4}$ gewählt, und man erhält drei neue Beweisverpflichtungen:

1. $even(x),\ \ x \geq 0,\ \ s = 0,\ \ y = x\ \ \vdash$
   $even(x) \wedge x \geq 0 \wedge \frac{y(y+2)}{4} = s + \frac{x(x+2)}{4}$ (I gilt am Anfang)

2. $even(x),\ \ x \geq 0,\ \ \frac{y(y+2)}{4} = s + \frac{x(x+2)}{4},\ \ x \neq 0\ \ \vdash$
   $[\texttt{s:=s+x; x:=x-2}]\ even(x) \wedge x \geq 0 \wedge \frac{y(y+2)}{4} = s + \frac{x(x+2)}{4}$ (I ist Invariante)

3. $even(x),\ \ x \geq 0,\ \ \frac{y(y+2)}{4} = s + \frac{x(x+2)}{4},\ \ \neg x \neq 0\ \ \vdash\ \ s = \frac{y(y+2)}{4}$

Die erste Beweisverpflichtung ist eine gültige Formel der Prädikatenlogik über einer arithmetischen Struktur und somit ein Axiom im Kalkül.
Die zweite Beweisverpflichtung wird durch zweimalige Anwendung der Regel (Zuweisung rechts) (R16) umgeformt zu

$$even(x'),\ \ x' \geq 0,\ \ \tfrac{y(y+2)}{4} = s' + \tfrac{x'(x'+2)}{4},\ \ x' \neq 0,\ \ s = s' + x',\ \ x = x' - 2\ \ \vdash$$
$$even(x) \wedge x \geq 0 \wedge \tfrac{y(y+2)}{4} = s + \tfrac{x(x+2)}{4}.$$

Dies läßt sich umformen und vereinfachen zu

$$even(x),\ \ x \geq 0,\ \ \tfrac{y(y+2)}{4} = s' + \tfrac{x'(x'+2)}{4},\ \ s = s' + x',\ \ x = x' - 2\ \ \vdash$$
$$even(x) \wedge x \geq 0 \wedge \tfrac{y(y+2)}{4} = s + \tfrac{x(x+2)}{4}.$$

und weiter zu

$$even(x),\ \ x \geq 0,\ \ \frac{y(y+2)}{4} = s - (x+2) + \frac{(x+2)(x+4)}{4}\ \ \vdash$$
$$even(x) \wedge x \geq 0 \wedge \frac{y(y+2)}{4} = s + \frac{x(x+2)}{4}.$$

Da gilt $s - (x+2) + \frac{(x+2)(x+4)}{4} = s + \frac{x(x+2)}{4}$, ist auch diese Sequenz gültig. Offensichtlich ist auch die dritte Beweisverpflichtung erfüllt und somit ist der Beweis geschlossen. Damit ist nun tatsächlich bewiesen, daß obiger Algorithmus für positive gerade Eingaben x die Summe der geraden Zahlen von Null bis x berechnet.
Nun sei angenommen, daß es nicht möglich ist, die geeignete Schleifeninvariante zu finden oder daß obiger Beweis für einen anderen Algorithmus zu aufwendig ist. Man will aber trotzdem sicher sein, daß zumindest während der ganzen Ausführung für alle positiven Eingaben x gilt: $x \geq 0$. Damit ist zwar noch nicht die Korrektheit des Algorithmus gezeigt, aber man kann zumindest sicher sein, daß die while-Schleife nicht aus dem Grunde, daß x kleiner als Null wird, nicht terminiert. Will man nun die Behauptung beweisen, beginnt man mit der Sequenz

$$x \geq 0\ \ \vdash$$
$$\subset\texttt{s:=0}\supset\subset\texttt{IF even(x) THEN WHILE x} \neq \texttt{0 DO s:=s+x; x:=x-2}\supset x \geq 0$$

Im ersten Ableitungsschritt erhält man durch Anwendung der Regel (Zuweisung rechts) (R34) zwei neue Beweisverpflichtungen, wovon die erste trivialerweise gültig ist:

1. $x \geq 0\ \ \vdash\ \ x \geq 0$
2. $x \geq 0,\ \ s = 0\ \ \vdash$
   $\subset\texttt{IF even(x) THEN WHILE x} \neq \texttt{0 DO s:=s+x; x:=x-2}\supset x \geq 0$

Im zweiten Ableitungsschritt wird die Regeln (if-then-else rechts) (R38) angewendet und man erhält die beiden Sequenzen

$$x \geq 0,\ \ s = 0,\ \ \neg even(x)\ \ \vdash\ \ x \geq 0.$$

und

$$x \geq 0,\ \ s = 0,\ \ even(x)\ \ \vdash \subset\texttt{WHILE x} \neq \texttt{0 DO s:=s+x; x:=x-2}\supset x \geq 0.$$

Es ist sofort ersichtlich, daß die erste Sequenz gültig ist. Um die while-Schleife in der zweiten Sequenz zu beseitigen, genügt nun die im Vergleich zu obigem Beweis einfache Invariante $x \geq 0 \wedge even(x)$. Die Anwendung der Regel (while rechts (Invarianz)) (R44) führt zu vier neuen Sequenzen.

1. $x \geq 0,\ \ s = 0,\ \ even(x)\ \ \vdash\ \ x \geq 0,\ \ even(x)$
2. $x \geq 0,\ \ even(x),\ \ x \neq 0\ \ \vdash\ \ [\texttt{s:=s+x; x:=x-2}]x \geq 0,\ \ even(x)$

3. $x \geq 0,\ \ even(x),\ \ x \neq 0\ \ \vdash\ \ \subset$`s:=s+x; x:=x-2`$\supset x \geq 0$

4. $x \geq 0,\ \ even(x),\ \ \neg x \neq 0\ \ \vdash\ \ x \geq 0$

Die erste und die vierte Sequenz sind offensichtlich gültig. Aus der zweiten erhält man durch zweimalige Anwendung der Regel (Zuweisung rechts) (R16) die Beweisverpflichtung

$$x' \geq 0,\ \ even(x'),\ \ x' \neq 0,\ \ s = s' + x',\ \ x = x' - 2\ \ \vdash\ \ x \geq 0,\ \ even(x),$$

die vereinfacht wird zu der offensichtlich gültigen Sequenz

$$x' \geq 2,\ \ even(x'),\ \ s = s' + x',\ \ x = x' - 2\ \ \vdash\ \ x \geq 0,\ \ even(x).$$

Bleibt schließlich noch die Gültigkeit der dritten Beweisverpflichtung zu zeigen. Man erhält durch zweimalige Anwendung der Regel (Zuweisung rechts) (R34) die drei Sequenzen

1. $x \geq 0,\ \ even(x),\ \ x \neq 0\ \ \vdash\ \ x \geq 0$

2. $x \geq 0,\ \ even(x),\ \ x \neq 0,\ \ s = s' + x\ \ \vdash\ \ x \geq 0$

3. $x' \geq 0,\ \ even(x'),\ \ x' \neq 0,\ \ s = s' + x',\ \ x = x' - 2\ \ \vdash\ \ x \geq 0$

Die ersten beiden sind wieder trivial. Die dritte wird vereinfacht zu

$$x' \geq 2,\ \ even(x'),\ \ s = s' + x',\ \ x = x' - 2\ \ \vdash\ \ x \geq 0.$$

Auch diese Sequenz ist gültig. Damit hat man bewiesen, daß tatsächlich für alle Eingaben x, die größer oder gleich Null sind, während der ganzen Ausführung gilt: $x \geq 0$.

Der Beweis wäre wesentlich einfacher und übersichtlicher geworden, hätte man den Algorithmus vereinfacht zu:

```
IF even(x) THEN
  WHILE x ≠ 0 DO
    x:=x-2
```

Dies wäre durchaus erlaubt, da man mit dieser Vereinfachung die Variable x nicht beeinflußt. Die Zeilen, die weggelassen wurden, dienten lediglich der Berechnung der Summe, die aber bei diesem Beweis nicht von Interesse war.

Eine weitere Anwendungsmöglichkeit sind Programme, die überhaupt nicht terminieren. Man stelle sich zum Beispiel ein Betriebssystem vor, das natürlich solange läuft wie der Rechner. In diesem Fall kann man mit der gewöhnlichen Dynamischen Logik nicht sehr viel anfangen, denn Formeln, die ein Programm enthalten,

das nicht terminiert, sind bei Verwendung der Modalität $[\cdot]$ trivialerweise immer gültig. Damit ist es also nicht möglich, zu beweisen, daß zum Beispiel ein Puffer niemals überläuft oder bestimmte Zählvariablen immer im gewünschten Bereich bleiben und noch vieles mehr.
Als Beispiel diene folgendes Programm:

```
y:=1
WHILE true DO
  IF y=1 THEN
    x:=x+1
    IF x=2 THEN y:=0
  ELSE
    x:=x-1
    IF x=0 THEN y:=1
```

Gilt im Anfangszustand $x = 0$, dann zählt dieser Algorithmus die Variable x endlos von Null hoch bis zum Wert zwei und wieder herunter auf Null. Um die Aussage zu beweisen, daß ständig $0 \leq x \leq 2$ gilt, ist die Modalität $\subset \cdot \supset$ erforderlich und man beginnt mit der Sequenz

$$x = 0 \;\vdash\; \subset\texttt{y:=1; WHILE } \mathit{true} \texttt{ DO IF y=1 THEN } \alpha \texttt{ ELSE } \beta\supset 0 \leq x \leq 2,$$

mit den Abkürzungen

$$\alpha := (\texttt{x:=x+1; IF x=2 THEN y:=0})$$
$$\beta := (\texttt{x:=x-1; IF x=0 THEN y:=1}).$$

Durch Anwendung der Regel (Zuweisung rechts) (R34) gelangt man zu den beiden Sequenzen

1. $x = 0 \;\vdash\; 0 \leq x \leq 2$
2. $x = 0,\; y = 1 \;\vdash\; \subset\texttt{WHILE } \mathit{true} \texttt{ DO IF y=1 THEN } \alpha \texttt{ ELSE } \beta\supset 0 \leq x \leq 2$

Die erste Sequenz ist offenbar gültig.
Um im nächsten Schritt die while-Schleife zu beseitigen, wird die Regel (while rechts (Invarianz)) (R44) benutzt. Als Schleifeninvariante wird $I = 0 \leq y \leq 1 \wedge (y = 0 \rightarrow x = 1 \vee x = 2) \wedge (y = 1 \rightarrow x = 0 \vee x = 1)$ gewählt. Man erhält so vier neue Beweisverpflichtungen:

1. $x = 0,\; y = 1 \;\vdash\; I$
2. $I,\; \mathit{true} \;\vdash\; [\texttt{IF y=1 THEN } \alpha \texttt{ ELSE } \beta]I$

3. $I, \ true \ \vdash \ \subset$`IF y=1 THEN` α `ELSE` $\beta\supset 0 \leq x \leq 2$

4. $I, \ \neg true \ \vdash \ 0 \leq x \leq 2$

Die Sequenzen eins und vier sind offensichtlich gültig.
Aus der zweiten Sequenz erhält man durch Anwendung der Regel (if-then-else rechts) (R20)

$$I, \ true, \ y = 1 \ \vdash \ [\alpha]I$$

und

$$I, \ true, \ \neg y = 1 \ \vdash \ [\beta]I.$$

Diese beiden Sequenzen lassen sich vereinfachen zu

$$(x = 0 \vee x = 1), \ y = 1 \ \vdash \ [\alpha]I$$

und

$$(x = 1 \vee x = 2), \ y = 0 \ \vdash \ [\beta]I.$$

Setzt man für α und β die konkreten Programme ein, sehen die beiden Sequenzen wie folgt aus:

1. $(x = 0 \vee x = 1), \ y = 1 \ \vdash \ [$`x:=x+1; IF x=2 THEN y:=0`$]I$

2. $(x = 1 \vee x = 2), \ y = 0 \ \vdash \ [$`x:=x-1; IF x=0 THEN y:=1`$]I$

Aus Sequenz 1 erhält man durch Beseitigen des Programms mit den Regeln (Zuweisung rechts) (R16) und (if-then-else rechts) (R20)

$$(x' = 0 \vee x' = 1), \ y' = 1, \ x = x' + 1, \ x = 2, \ y = 0 \ \vdash$$
$$0 \leq y \leq 1 \wedge (y = 0 \rightarrow x = 1 \vee x = 2) \wedge (y = 1 \rightarrow x = 0 \vee x = 1)$$

und

$$(x' = 0 \vee x' = 1), \ y = 1, \ x = x' + 1, \ \neg x = 2 \ \vdash$$
$$0 \leq y \leq 1 \wedge (y = 0 \rightarrow x = 1 \vee x = 2) \wedge (y = 1 \rightarrow x = 0 \vee x = 1),$$

welche beide offensichtlich gültig sind.
Analog erhält man aus Sequenz 2 die beiden ebenfalls gültigen Sequenzen

$$(x' = 1 \vee x' = 2), \ y' = 0, \ x = x' - 1, \ x = 0, \ y = 1 \ \vdash$$
$$0 \leq y \leq 1 \wedge (y = 0 \rightarrow x = 1 \vee x = 2) \wedge (y = 1 \rightarrow x = 0 \vee x = 1)$$

und

$$(x' = 1 \vee x' = 2), \;\; y = 0, \;\; x = x' - 1, \;\; \neg x = 0 \;\; \vdash$$
$$0 \leq y \leq 1 \wedge (y = 0 \rightarrow x = 1 \vee x = 2) \wedge (y = 1 \rightarrow x = 0 \vee x = 1).$$

Um den Beweis zu schließen, ist noch die Gültigkeit der Sequenz 3

$$I, \;\; \mathit{true} \;\; \vdash \;\; \subset\texttt{IF y=1 THEN } \alpha \texttt{ ELSE } \beta \supset 0 \leq x \leq 2$$

zu zeigen. Diese wird vereinfacht und durch Anwendung der Regel (if-then-else rechts) (R38) erhält man die zwei neuen Beweisverpflichtungen

$$(x = 0 \vee x = 1), \;\; y = 1 \;\; \vdash \;\; \subset\texttt{x:=x+1; IF x=2 THEN y:=0}\supset 0 \leq x \leq 2$$

und

$$(x = 1 \vee x = 2), \;\; y = 0 \;\; \vdash \;\; \subset\texttt{x:=x-1; IF x=0 THEN y:=1}\supset 0 \leq x \leq 2.$$

Die Ableitung dieser beiden Sequenzen soll aber hier nicht mehr vorgeführt werden, da man sich erstens leicht davon überzeugen kann, daß diese tatsächlich gelten, und zweitens die Ableitung fast analog zur obigen verläuft und ziemlich unübersichtlich wird, da eine Menge neuer Beweisverpflichtungen generiert werden.

9 Diskussion

In dieser Studienarbeit wurde im ersten Teil zuerst ein Überblick über die Dynamische Aussagenlogik und die Dynamische Quantorenlogik gegeben. Diese wurden dann um neue Modalitäten zur Prozeßaussagenlogik bzw. Prozeßlogik erster Stufe hin erweitert. Das Ziel war schließlich, für die Prozeßlogik erster Stufe einen Sequenzenkalkül zu entwickeln. Die Korrektheit des Kalküls wurde aus Platz- und Zeitgründen nur exemplarisch an zwei Regeln demonstriert, wohingegen die (relative) Vollständigkeit mit Hilfe des Vollständigkeitstheorems von Harel bewiesen wurde.
Zum Ende hin wurde der Kalkül schließlich an zwei Beispielen demonstriert. Betrachtet man die einfachen Beispiele und die doch recht aufwendigen Ableitungen, so wird einem schnell klar, daß ein solcher Kalkül ohne Rechnerunterstützung ziemlich wertlos ist, insbesondere wenn man sich einen Kalkül für eine „echte“ Programmiersprache, wie zum Beispiel Java, vorstellt. Man sieht allerdings auch, daß die Unterstützung eines Rechners sich in erster Linie darauf beschränken wird, einfache Ableitungsschritte *fehlerfrei* durchzuführen, was dem Menschen nicht unbedingt möglich ist, da er sehr schnell den Überblick verliert und Flüchtigkeitsfehler begeht. Dennoch ist ein automatischer Beweis ohne Interaktion eines Menschen nur in den wenigsten Fällen vorstellbar, denn geht es zum Beispiel darum, Invarianten zu finden, ist die Kreativität des Menschen noch immer unverzichtbar.

Literatur

[ABBG+00] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Wolfram Menzel und Peter H. Schmitt. The KeY Approach: Integrating Object Oriented Design and Formal Verification. In Gerhard Brewka und Luís Moniz Pereira (Hrsg.), *Proc. 8th European Workshop on Logics in AI (JELIA)*, LNCS. Springer-Verlag, Oktober 2000. URL: `ftp://ftp.cs.chalmers.se/pub/users/reiner/jelia.ps.gz`.

[Burg84] John P. Burgess. Basic Tense Logic. In D. Gabbay und F. Guenthner (Hrsg.), *Handbook of Philosophical Logic, Volume II: Extensions of Classical Logic*, Band 165, Kapitel II.2, S. 89–133. Dordrecht, 1984.

[BuSe84] Robert Bull und Krister Segerberg. Basic Modal Logic. In D. Gabbay und F. Guenthner (Hrsg.), *Handbook of Philosophical Logic, Volume II: Extensions of Classical Logic*, Band 165, Kapitel II.1, S. 1–88. Reidel, Dordrecht, 1984.

[Chel80] Brian F. Chellas. *Modal Logic: An Introduction.* 1980.

[Enge67] E. Engeler. *Algorithmic properties of structures*, S. 183–195. Math. Systems Theory. 1967.

[Floy67] R. W. Floyd. Assigning meanings to programs. In *Proc. AMS Symp. on Applied Mathematics 19.* Amer. Mathematical Soc., 1967, S. 19–31.

[Gent35] G. Gentzen. Untersuchungen über das Logische Schliessen. *Math. Zeitschr., 39*, 1934-1935, S. 176–210.

[HaKo82] David Harel und Dexter Kozen. Process Logic: Expressiveness, Decideability, Completeness. *Journal Of Computer And System Sciences 25*, 1982, S. 144–170.

[Hare79] David Harel. *First Order Dynamic Logic.* Lecture Notes in Computer Science 68. Goos and Hartmanis. 1979.

[Hare84] David Harel. Dynamic Logic. In D. Gabbay und F. Guenthner (Hrsg.), *Handbook of Philosophical Logic, Volume II: Extensions of Classical Logic*, Kapitel II.10, S. 497–604. Dordrecht, 1984.

[Hoar67] C. A. R. Hoare. An axiomatic basis for computer programming. In *Comm. ACM 12*, 1967, S. 516–580.

[KoTi90] Dexter Kozen und Jerzy Tiuryn. Logic of Programs. In J. van Leeuwen (Hrsg.), *Handbook of Theoretical Computer Science*, Band 165, Kapitel 14, S. 89–133. Elsevier Science Publishers B.V., Dordrecht, 1990.

[Lemm77] Edward John Lemmon. *An Introduction to Modal Logic (The "Lemmon Notes")*, Band 11 der *Americal Philosophical Quarterly Monograph Series*. Basil Blackwell, Oxford. 1977.

[Pnue77] A. Pnueli. The temporal logic of programs. In *Proc. 18th FOCS*, October 1977, S. 46–57.

[Pnue79] A. Pnueli. The temporal semantics of concurrent programs. In *Proc. Int. Symp. on Semantics of Concurrent Programs*, Evian, France, July 1979.

[Prat77] V. R. Pratt. Semantical considerations on Floyd-Hoare logic. In *Proc. 18th Ann. IEEE Symp. on Foundation of Computer Science*, 1977, S. 109–121.

[Prat79] V. R. Pratt. Process Logic: Preliminary Report. In *6th ACM Symposium on Principles of Programming Languages, San Antonio, Texas*, 1979.

[Reif92] W. Reif. The KIV System: Systematic Construction of Verified Software. In D. Kapur (Hrsg.), *11th Conference on Automated Deduction. Proceedings*, Lecture Notes in Computer Science. Albany, NY, USA, Springer, 1992.

[Schm99] P. H. Schmitt. Skriptum zur Vorlesung „Nichtklassische Logiken", 1999.

[Thie66] H. Thiele. *Wissenschaftstheoretische Untersuchungen in Algorithmischen Sprachen. Theorie der Graphschemata-Kalküle*. VEB Deutscher Verlag der Wissenschaften, Berlin, DDR. 1966.