Automatische Erzeugung von Verifikations- und Falsifikationsbedingungen sequentieller Programme

D i s s e r t a t i o n

zur Erlangung des akademischen Grades

doctor rerum naturalium (Dr. rer. nat.)

vorgelegt dem Rat der Fakultät für Mathematik und Informatik
der Friedrich-Schiller-Universität Jena.

von Dipl.-Inf. Stefan Kauer

geboren am 22.4.1964 in Fulda

Gutachter

1. Prof. Dr. Jürgen F. H. Winkler, Jena
2. Prof. Dr. Eike Best, Oldenburg
3. Prof. Dr. Gerald Futschek, Wien

Tag des Rigorosums: 15. Januar 1998

Tag der öffentlichen Verteidigung: 27. Januar 1999

Inhalt

1 EINLEITUNG ... 3

1.1 ZUVERLÄSSIGKEIT VON SOFTWARE ... 4
1.2 DER FREGE PROGRAM PROVER FPP ... 4
1.3 AUFBAU DER ARBEIT ... 5
1.4 EINGRENZUNG DER ARBEIT ... 7
1.4.1 Programmiersprache ... 7
1.4.2 Erläuterung des Problems der Programmverifikation ... 7

2 NOTATIONEN UND BEGRIFFE ... 10

2.1 GRUNDLEGENDE DEFINITIONEN UND SÄTZE ... 12
2.2 RELATIONALE SEMANTIK ... 15
2.2.1 Programmzustände ... 16
2.2.2 Korrektheit ... 18
2.2.3 Semantische Äquivalenz ... 21
2.3 EIGENSCHAFTEN VON *WP* ... 22

3 VERIFIKATIONSBEDINGUNGEN ... 30

3.1 ÜBERBLICK UND BEGRIFFSBESTIMMUNG ... 30
3.2 EXAKTE VERIFIKATIONSBEDINGUNGEN ... 31
3.2.1 Null-Statement ... 31
3.2.2 Zuweisung ... 32
3.2.3 Sequenz ... 37
3.2.4 Fallunterscheidung ... 38
3.2.4.1 If-Statement ... 38
3.2.4.2 Case-Statement ... 40
3.2.5 While-Schleifen ... 42
3.2.6 Nichtdeterministische Auswahl ... 47
3.3 SUBSTITUTIONSLEMMA FÜR *WP* ... 48
3.4 APPROXIMIERTE VERIFIKATIONSBEDINGUNGEN ... 52
3.4.1 While-Schleifen ... 52
3.4.2 For-Schleifen ... 63
3.4.2.1 Aufwärtszählende For-Schleifen ... 63
3.4.2.2 Existenz der Invarianten ... 66
3.4.2.3 Abwärtszählende For-Schleifen ... 67
3.4.2.4 Beispiele ... 67
3.4.3 Prozeduraufrufe ... 69
3.4.3.1 Elimination von Funktionen und globalen Variablen ... 71
3.4.3.2 Die schwächst mögliche Vorbedingung eines Prozeduraufrufs ... 76
3.4.4 Eine Vorbedingung für Schleifen im Kontext einer umgebenden Anweisung ... 83
3.4.5 Rekursive Prozeduren ... 86

4 BEWEIS VON SCHLEIFEN OHNE INVARIANTE ... 96

4.1 ERZEUGUNG VON INVARIANTEN FÜR FOR-SCHLEIFEN ... 96
4.2 BERECHNUNG DER *WP* VON WHILE-SCHLEIFEN ... 106
4.2.1 Vorhandene Methoden zur Ermittlung einer Invarianten ... 106
4.2.2 Automatische Konstruktion von μf ... 112
4.2.3 Klassifikation des Problems ... 116
4.2.4 Überblick über das Verfahren ... 118
4.2.5 Transformation der Terme und Reduktion auf Unifikation ... 120
4.2.6 Die Generierung der Formeln aus der gegebenen Sequenz ... 126
4.2.7 Behandlung von Sequenzen nicht strukturgleicher Prädikate ... 130
4.2.7.1 Transformation in strukturgleiche Prädikate ... 130
4.2.7.2 Zerlegung in Teilmengen mit Indizes strukturgleicher Konjunkte ... 131
4.2.7.3 Beispiele ... 138

5 FALSIFIKATIONSBEDINGUNGEN ... 145

5.1 FORMALISIERUNG DES PROBLEMS ... 147
5.2 REPRÄSENTATION VON PRÄDIKATEN UND FUNKTIONEN ... 149
5.3 AFFINISIERUNG BESTIMMTER POLYNOMIALER CONSTRAINTS ... 149

5.3.1 Faktorisierung *150*
5.3.2 Affinisierung *150*
5.4 UNINTERPRETIERTE FUNKTIONSSYMBOLE 151
5.5 ELIMINIERUNG INNERER QUANTOREN 151
5.6 ELIMINIERUNG VON ∨ 152
5.7 LOKALISIERUNG EINER VERLETZTEN ZUSICHERUNG 153
5.8 BEISPIELE 153
AUSBLICK 165
LITERATUR 167
STICHWORTVERZEICHNIS 174

1 Einleitung

In den letzten Jahren hat sich immer mehr die Erkenntnis durchgesetzt, daß die Erstellung von Software eine ingenieurmäßige Aufgabe ist. Man muß also versuchen, Vorgehensweisen der klassischen Ingenieurdisziplinen auf die Softwareentwicklung zu übertragen. Für viele Softwareentwickler ist die Erstellung von Software noch so etwas wie eine Kunst, was sich auch im Titel des klassischen Werks von D. Knuth *The Art of Computer Programming* [K97] widerspiegelt. Das hat Parallelen zu klassischen Ingenieurdisziplinen wie Hochbau oder Maschinenbau, die am Anfang ihrer Entwicklung auch mehr Kunst als Wissenschaft waren und damit an bestimmte Personen gebunden waren, die eben zu gerade dieser Kunst talentiert waren. So gab es im ausgehenden Mittelalter nur wenige Baumeister, die eine Kathedrale bauen konnten, da das eben eine Kunst war. Oder im letzten Jahrhundert war es eine Kunst, eine Lokomotive zu bauen. Ein weiteres Beispiel ist das Schleifen von Linsen mit gewünschten Eigenschaften. Auch das war in der Zeit vor Ernst Abbe eine Kunst. Die Linsen wurden so lange geschliffen und ihre Eigenschaften ermittelt, bis sie die gewünschten Vorgaben erfüllten. Durch die Arbeit von Ernst Abbe wurde dieses (Kunst)Handwerk auf eine wissenschaftliche Grundlage gestellt. Auch er erkannte, daß man mit diesem sogenannten „Pröbeln“ bzw. handwerklicher Erfahrung nicht weit kommt [SW93].
Dieser Schritt von der Kunst zur Wissenschaft hat sich in den klassischen Ingenieurdisziplinen vollzogen. Auch in einer noch nicht so alten Disziplin vollzieht sich dieser Schritt, nämlich in der Entwicklung bestimmter pharmazeutischer Produkte. Dort werden Substanzen mit bestimmten, vorgegebenen Eigenschaften gesucht. Auch in dieser Disziplin wird viel mit Versuch und Irrtum gearbeitet. Doch in letzter Zeit ist es auch hier möglich, mehr ingenieurmäßig vorzugehen. Moleküle mit bestimmten Eigenschaften werden konstruiert, genau wie eine Maschine, was unter dem Schlagwort *drug design* [TA95], [RHC95] bekannt ist.
Die Softwareentwicklung nach Versuch und Irrtum könnte man damit vergleichen, eine Brükke zu bauen, die bei der geringsten Belastung einstürzt. Der Grund dafür könnten z.B. zu dünne Pfeiler sein. Also werden beim nächsten Versuch dickere Pfeiler verwendet u.s.w. bis die Brücke den Anforderungen genügt. Es wird jedoch so vorgegangen, daß vor dem Bau der Brücke ausgerechnet wird, wie viele und welche Pfeiler wo stehen müssen. Die Brücke wird dann mit einer hohen Last getestet, um sicher zu gehen, daß die Anforderungen erfüllt sind. In der Softwareentwicklung wird ein fertiges Programm mit einer Menge von Testfällen getestet und gegebenenfalls verbessert, bis alle diese Testfälle fehlerfrei abgearbeitet werden. Dann ist bewiesen, daß das Programm für alle Testfälle korrekt ist. Außerdem hat man durch geschickte Auswahl der Testfälle ein starkes Indiz für die Fehlerfreiheit des Programms, allerdings keinen Beweis. Ein erster Schritt zur Softwareerstellung auf wissenschaftlicher Grundlage ist die axiomatische Methode von Floyd [F67] und Hoare [H69]. Eine sehr gute Darstellung dieser Technik beinhaltet das Buch *The Science of Programming* von David Gries [G81], dessen Titel man auch als Anwort auf den Titel von Donald Knuths Buch verstehen kann. Weitere Bücher mit ähnlichem Inhalt sind z.B. [F89], [B90] und [Ba89].
Nach jahrzehntelanger, weltweiter Programmiererfahrung in prozeduralen Programmiersprachen, in denen der weitaus größte Teil aller Programme geschrieben wird, hat sich herausgestellt, daß gewisse Programmkonstrukte besonders fehleranfällig sind, z.B. unbedingte Sprungbefehle, Verweise oder globale Variablen. Und gerade diese unsicheren Konstrukte sind es, die bei der axiomatischen Methode schwierig zu formalisieren sind. Es gibt deshalb Richtlinien, solche Konstrukte für Software, die sehr hohen Sicherheitsanforderungen genügen muß, nicht zu verwenden [BWB92]. Diese Richtlinien haben zum Teil Gesetzescharakter wie auch z.B. Vorschriften zur Verwendung bestimmter Baumaterialien oder -techniken. Das Argument, daß dadurch der Programmierer in seiner Freiheit oder Kreativität eingeschränkt wird, kann man in diesem Zusammenhang nicht gelten lassen. Vielmehr wird man durch diese

Technik veranlaßt, das Problem tiefer zu durchdenken und sich nicht nur auf seine Intuition zu verlassen.

1.1 Zuverlässigkeit von Software

Über die Notwendigkeit formaler Methoden zur Unterstützung der Entwicklung komplexer (Programm)systeme ist man sich in der Softwaretechnik im Klaren. Das zeigen auch z.B. die Vorfälle mit Therac25 [LT93] oder Ariane5 [L96].
Für die Erstellung sicherer und zuverlässiger Software sind in den verschiedenen Phasen der Softwareentwicklung verschiedene Techniken und Methoden erforderlich [S92]. Eine dieser Techniken ist die formale Verifikation [G81]. Damit kann überprüft werden, ob ein gegebenes Programm eine formale Spezifikation erfüllt.
Damit ist aber noch nicht sichergestellt, daß das Programm den gewünschten Anforderungen entspricht, sondern nur, daß es die formale Spezifikation erfüllt. Es ist dann immer noch möglich und bei nicht trivialen Problemen wahrscheinlich, daß das Programm die Anforderungen nicht erfüllt. Das liegt dann daran, daß die am Anfang stehenden verbalen Anforderungen nicht korrekt in eine formale Spezifikation umgesetzt wurden. Dieses äußerst schwierige und zur Zeit ungelöste Problem ist nicht Gegenstand der vorliegenden Arbeit.
Eine Technik zur Überprüfung der Programmkorrektheit, d.h. der Übereinstimmung von Implementierung und formaler Spezifikation, ist die Bestimmung der schwächsten Vorbedingung bzw. weakest precondition. Diese Technik läßt sich aber nicht oder nur schwer auf alle Programmkonstrukte anwenden. Z.B. gibt es für Schleifen und Funktions- / Prozeduraufrufe andere Techniken (Schleifeninvariante). Gemeinsam ist ihnen, daß sie sehr aufwendig und daher fehleranfällig von Hand durchzuführen sind. Das ist ein Grund dafür, daß die formale Programmverifikation noch kaum verbreitet ist. Diese Techniken sind prinzipiell auf alle prozeduralen Sprachen anwendbar wie C, FORTRAN, COBOL, Pascal und Ada.
In dieser Arbeit wird davon ausgegangen, daß Spezifikation und Programm gegeben sind. Falls Spezifikation und Programm nicht übereinstimmen, bedeutet das, daß das Programm nicht korrekt ist und modifiziert werden muß.

1.2 Der Frege Program Prover FPP

Im Rahmen dieser Arbeit wurde das Tool FPP (*Frege program prover*) entwickelt, das die Verifikation weitgehend automatisieren soll. FPP ist ein experimentelles System zur automatischen Programmverifikation. FPP ermöglicht die Eingabe von einfachen Programmen in einer Teilsprache von Ada, die um Zusicherungen erweitert wurde. Eine Beschreibung der Arbeitsweise von FPP ist [K98]. FPP ist damit ähnlich z.B. zu Tatzelwurm [K94], KIV [HMRS90], Penelope [O94/1], [O94/2], [O94/3], [O94/4] und NPPV (New Paltz Program Verifier) [G97: 184-188]. Tatzelwurm und NPPV sind Verifikationssysteme für Teilsprachen von Pascal, Penelope ist ein Verifikationssytem für eine Teilsprache von Ada. Tatzelwurm und Penelope verwenden interaktive Beweiser, d.h. die Hauptlast des Beweises der *Verifikationsbedingungen (verification conditions VCs*) hat der Benutzer zu tragen. NPPV und FPP verwenden einen autonomen Beweiser, sind also vollautomatisch. Die Unterschiede der einzelnen Systeme liegen also hauptsächlich im verwendeten Beweiser. Gemeinsam ist ihnen der *verification condition generator* (VCG). Er generiert aus dem Programm mit den Zusicherungen die einzelnen VCs. Ein Vergleich von NPPV und FPP ist in [KW98].
FPP besteht im Wesentlichen aus einem in Ada geschriebenen VCG und einem automatischen Beweiser. Der VCG erstellt aus den Anweisungen des Programms und der formalen Spezifikation logische Implikationen, die vom Beweiser auf Gültigkeit überprüft werden.
Zu diesem Zweck wurden verschiedenen Beweiser daraufhin untersucht, ob und wie geeignet sie für diese Anwendung sind. Da das System autonom sein soll, scheidet ein interaktiver

Beweiser aus. In einem interaktiven Beweiser leitet der Benutzer den eigentlichen Beweisprozeß in dem Sinne, daß der Benutzer entscheidet, welcher Schritt als nächster gemacht wird. Der Beweiser führt dann diesen Schritt aus und führt eventuell noch andere Funktionen wie Fehlerprüfung und Protokollierung u.ä. durch. Ein Beispiel für ein solches System ist IMPS [FGT94]. Das Referenzmodell autonomer Beweiser ist Otter von W. McCune [M94]. Dieser Beweiser ist ein Beweiser für Prädikatenlogik 1. Ordnung. Da er ein rein logischer Beweiser ohne eingebaute Theorie ist, ist er für Beweise einer konkreten Theorie nicht geeignet. Ideal ist eine Kombination von logischem Beweiser mit symbolischer und numerischer Mathematik, der z.B. $3n + 4n$ gleich in $7n$ mit Hilfe von Computeralgebra-Algorithmen umformen kann, ohne einen Umweg über einen langwierigen und möglicherweise nicht zu bewältigenden Beweis gehen zu müssen (für ein umfangreicheres Beispiel siehe auch Beweis von Lemma 4.21). Als sehr geeignet hat sich Analytica von E.M. Clarke von der Carnegie Mellon University in Pittsburgh [CZ92] erwiesen. Es ist in Mathematica [W92] geschrieben, so daß es sich relativ leicht ändern bzw. ergänzen läßt. Es wurden einige Änderungen und Erweiterungen vorgenommen, um Analytica an die Anforderungen von FPP anzupassen. Die 3 wesentlichen Erweiterungen bestehen in einer Vorverarbeitung, Transformation auf Klauselform und Anwendung von Ersetzungsregeln. Das Ergebnis ist *mexana*. Über eine in C geschriebene Schnittstelle kann *mexana* vom VCG aus aufgerufen werden.
Da VCs im Allgemeinen recht groß werden können, d.h. sich über mehrere Zeilen erstrecken können, ist für eine lesbare Ausgabe ein Pretty-Printer erforderlich. Für FPP wurde eigens ein einfacher Pretty-Printer Printer für Ausdrücke mit Präfix- und Infix-Operatoren entwickelt. Zur übersichtlichen Gestaltung langer Formeln bzw. Beweise gibt es einige informelle Ideen von Lamport, die allerdings eher für den menschlichen Benutzer gedacht sind und nicht einfach zu automatisieren sind [L93], [L94]. In der funktionalen Programmierung treten auch häufig große Ausdrücke auf, diese liegen allerdings nur in Präfixform vor, weshalb man einen Pretty-Printer für solche Ausdrücke [H95] nicht einfach übernehmen kann, ebenso wenig wie einen Pretty-Printer für Programme [R97]. Die VCs sind Ausdrücke, die näher an den Ausdrücken der täglichen Mathematik sind, d.h. sie enthalten Präfix- und Infix-Operatoren. Zu den Präfixoperatoren zählen auch die Array-Bezeichner. Es nutzt auch nichts, die Infix-Teilausdrücke in äquivalente Präfix-Teilausdrücke zu transformieren, den Pretty-Printer für Präfix-Ausdrücke darauf anzuwenden und dann zurück zu transformieren.
FPP ist auf allen HW-Plattformen und Betriebssystemen lauffähig, auf denen Ada, C und Mathematica existieren, damit z.B. auf Unix-Varianten und WindowsNT. Das zugrunde liegende Betriebsystem sollte allerdings multitaskingfähig sein. FPP ist außerdem über WWW unter der Adresse `http://www1.informatik.uni-jena.de/FPP/FPP-main.htm` als interaktive Anwendung verfügbar. Ein kleines FPP-Beispiel befindet sich in Abschnitt 3.2.4.1.
Mit FPP kann der Zeitaufwand für die formale Programmverifikation erheblich verringert werden und, was noch wichtiger ist, die Fehleranfälligkeit stark reduziert werden. Damit müßte es möglich sein, die formale Programmverifikation in der industriellen Praxis und der Lehre weiter zu verbreiten und somit einen Schritt in Richtung sichere Software zu gehen.

1.3 Aufbau der Arbeit

Das Ziel der Arbeit ist unter anderem eine möglichst weitgehende Automatisierung der Verifikation. Das bedeutet, daß alle in dieser Arbeit vorgestellten Verfahren rein mechanisch sind und ohne menschliches Eingreifen durchgeführt werden können. Zunächst folgt eine Eingrenzung der Arbeit mit Erwähnung der behandelten Programmkonstrukte.
Das 2. Kapitel beinhaltet die Erläuterung der verwendeten Notationen für Formeln, Beweise usw. sowie grundlegende Definitionen und Sätze, soweit sie in dieser Arbeit benötigt werden, darunter auch die formale Definition des hier verwendeten Begriffes der Korrektheit. Daneben werden bestimmte Eigenschaften der *wp*-Funktion nicht mit Hilfe der Prädikatenlogik,

sondern mit Hilfe der Mengenlehre bewiesen. Dazu werden statt Prädikaten die Mengen betrachtet, deren charakteristisches Prädikat sie sind. Es zeigt sich, daß die Beweise auf diese Weise einfacher und kürzer werden.
Das 3. Kapitel behandelt die Beweisregeln bzw. *verification conditions* (*VCs*) für verschiedene Programmkonstrukte. Die Semantik eines Programms wird dabei nicht durch seine schwächste Vorbedingung definiert, sondern auf eine eher operationelle Weise durch seine Programmfunktion. Die *wp*-Regel einer Anweisung wird dann als Satz formuliert und mit Hilfe der zu dieser Anweisung gehörenden Programmfunktion hergeleitet. Mit Hilfe der Programmfunktion ist es leichter möglich, eine verbal gegebene Semantikbeschreibung zu formalisieren. Neu ist hier nicht das Ergebnis, nämlich die altbekannten *wp*-Regeln, sondern eine einleuchtendere Weise ihrer Herleitung.
Es wird explizit die Äquivalenz der Existenz von Invariante und Terminierungsfunktion und der totalen Korrektheit einer Schleife gezeigt. Es folgen in der Literatur bisher nicht behandelte VCs:

- eine explizite Typprüfung bei der Zuweisung
- eine leistungsfähigere VC für Schleifen im Kontext umgebender Anweisungen
- eine neue VC für For-Schleifen
- eine leistungsfähigere VC für rekursive Prozeduren

Ein besonderer Schwerpunkt bildet die Behandlung von Schleifen im 4. Kapitel. Es wird erstmals eine Technik zur automatischen Ermittlung einer Invarianten für For-Schleifen vorgestellt. Den Hauptteil bildet ein neuartiges Verfahren zur Ermittlung der schwächsten Vorbedingung von Schleifen ohne Angabe von Invariante und Terminierungsfunktion.
Da während der Programmentwicklung im Allgemeinen die Programme nicht der Spezifikation genügen, ist es wichtig, ein Gegenbeispiel zu finden, das zum einen die Nichtkorrektheit explizit zeigt und zum anderen einen nützlichen Hinweis zur Verbesserung gibt. Das ist der Gegenstand des 5. Kapitels, die automatische Generierung eines Gegenbeispiels oder einer *falsification condition* FC zu einer nicht beweisbaren VC. Dies ist ein bezüglich Benutzerfreundlichkeit sehr nützlicher Teil eines automatischen Verifikationssytems.
Dabei stellt sich heraus, daß sich dieses Problem wie das Problem der automatischen Datenabhängigkeitsanalyse auf ein *constraint programming problem* cpp reduzieren läßt. Daher lassen sich gewisse Techniken aus diesem Bereich auf die Generierung von FCs anwenden, was hier erstmals getan wurde. Als besonders geeignet haben sich in diesem Zusammenhang die Verfahren von Pugh und Wonnacott erwiesen. Zunächst wird die nicht beweisbare VC in ein cpp transformiert. Dann wird versucht, dieses in ein äquivalentes *linear integer programming problem* (lipp) zu transformieren, das mit Hilfe des Ω-Tests [P92] eventuell gelöst wird. Die Transformationen der FC auf ein lipp gehen im Wesentlichen auch auf verschiedene Arbeiten von Pugh und Wonnacott zurück. Zudem wird gezeigt, daß auch eine VC in ein cpp transformierbar ist, das aber relativ nutzlos ist, da die VC genau dann gilt, wenn das zugehörige cpp keine Lösung hat. Da die Lösbarkeit nur partiell entscheidbar ist, kann man damit nichts zum Beweis der VC beitragen, da man nicht mehr auf der sicheren Seite ist.
Die Kapitel 3, 4 und 5 sowie die Abschnitte 3.4.2, 3.4.3, 3.4.5 und 4.2 werden mit einem groben Überblick eingeleitet und enthalten für den Fall, daß es zu diesem Gebiet ähnliche Arbeiten gibt, einen kurzen Vergleich damit.
Einen wesentlichen Teil der Arbeit machen Beispiele aus. Diese wurden unter folgenden Aspekten ausgewählt:

- sehr einfache Beispiele zum „Einstieg“ in ein Verfahren
- kleinere „echte“ Algorithmen
- Zusammenspiel verschiedener Gesichtspunkte
- exemplarisch und illustrativ für eine große Klasse von typischen Fällen

Daraus ergibt sich, daß Beispiele einen relativ großen Raum einnehmen, wie auch in allen Büchern und andern Veröffentlichungen zum Thema Programmverifikation. Das liegt einfach in der Natur dieses Themas.
An dieser Stelle möchte ich meinem Betreuer, Herrn Prof. Dr. Jürgen F. H. Winkler, für viele hilfreiche Gespräche und Anregungen danken. Nicht zu vergessen sind meine Kollegen am Institut für Informatik, insbesondere Herr Dr. Sebastian Schmidt für seine Tips für die C-Programmierung und Herr Dr. Gregor Weske für die Administration der Hard- und Software.

1.4 Eingrenzung der Arbeit

1.4.1 Programmiersprache

Als Programmiersprache wird eine Teilmenge von Ada [TD95] verwendet. Ada ist eine realistische Programmiersprache, und nicht eine zum Zweck der Verifikation speziell entwickelte Notation. Die betrachteten Datentypen sind `boolean` und `integer` sowie Teilbereichstypen davon und darauf aufbauende Arrays und Records. Der Datentyp `integer` wird als endliches Intervall der ganzen Zahlen betrachtet. Die in der Verifikation und überhaupt in der Programmierung häufig gemachten Annahmen `integer` = $\mathbb{Z}$ und `float` = $\mathbb{R}$ führen zu einfachen, aber nicht korrekten *wp*-Regeln bei der Wertzuweisung [H93: 36]. $\mathbb{Z}$ mit den 4 arithmetischen Grundoperationen ist eine andere algebraische Struktur als `integer` mit den 4 arithmetischen Grundoperationen. Der Unterschied liegt in der der Theorie zu Grunde liegenden Axiomenmenge. Der Unterschied zwischen `integer` und $\mathbb{Z}$ ist noch relativ gering; er erschöpft sich in der Endlichkeit von `integer`. Der Datentyp `float` unterscheidet sich von $\mathbb{R}$ nicht nur durch die Endlichkeit des Bereiches, sondern auch noch durch einige andere Merkmale, wie z.B. verschiedene Implementierungen, Modellzahlen, Rundung, Exceptions u.ä. Das liegt außerhalb des Bereiches der vorliegenden Arbeit. Solche Fragen werden teilweise in [IEEE85] und [EP97] behandelt.
Die Programmverifikation beruht auf einer statischen Analyse des Programmtextes. Durch die Verwendung von Pointern kann man dynamische Datenstrukturen erstellen, deren Analyse zur Compilezeit sehr schwierig ist. Die *wp*-Regel für die Zuweisung beruht im Wesentlichen darauf, daß nur der Wert derjenigen Variable geändert wird, die auf der linken Seite der Zuweisung steht. Durch die Verwendung von Pointern kann Aliasing auftreten. Dadurch können durch eine Zuweisung auch noch andere als die direkt betroffene Variable geändert werden. Damit ist die *wp*-Regel nicht mehr korrekt. Aliasing kann zwar auch bei Arrays auftreten, aber wegen der relativ einfachen Struktur von Arrays kann man dieses Problem durch eine Fallunterscheidung in den Griff bekommen. Es gibt wenige Ansätze zur Behandlung von Pointern mit *wp*-Regeln ([B89], [LS79], [JJKS97], [M93]). Gemeinsam ist ihnen eine relativ hohe Komplexität, Einschränkung der dynamischen Datenstrukturen (z.B. nur einfach verkettete Listen) und Einschränkungen der Ausdrucksfähigkeit der Zusicherungen.
Die in der Arbeit betrachteten Anweisungen sind das Null-Statement, die Zuweisung zu einer Variablen oder einem einzelnen Array- oder Record-Element, das If- und Case-Statement, die For- und While-Schleife sowie Prozeduraufrufe.

1.4.2 Erläuterung des Problems der Programmverifikation

Mit Hilfe zweier kleiner Beispiele soll das Problem der Programmverifikation kurz erläutert werden und gegen die Methode der Laufzeitchecks abgegrenzt werden. Zunächst wird der grobe Ablauf eines Verifikationssystems `verif` dargestellt. Dabei wird schon auf die in Kapitel 5 behandelten Falsifikationsbedingungen FC zurückgegriffen (eine kurze Erläuterung zu FCs wurde auch schon in Abschnitt 0 gegeben).
Die Eingabe besteht aus einem Programm mit Zusicherungen.

```
          function verif(P: Program_with_assertions) return re-
sult_type is
          begin
            generate VCs;
            if all VCs are provable
          then return correct;
            else  generate FC;
                  if FC solvable
                then  generate counterexample;
                        return (not_correct, counterexample);
                  else return not_decidable;
                  end if;
            end if;
          end verif;
```

D.h., daß es vorkommen kann, daß ein Programm weder bewiesen noch widerlegt werden kann. Das liegt einfach daran, daß sowohl das Verfahren zum Beweisen der VC als auch das Verfahren zur Lösung der FC begrenzt ist.

Nun folgt eine kurze Gegenüberstellung mit Laufzeitchecks. Die Korrektheit eines Programms soll durch Prüfung der Gültigkeit von Zusicherungen sichergestellt werden.

Beispiel:

```
x := a;
y := x + 1;
-- y = a+1
```

Das Programm besteht aus den beiden Anweisungen, die Zusicherung ist die prädikatenlogische Aussage $y = a + 1$. Das bedeutet, daß nach Ausführung der beiden Anweisungen immer die Zusicherung gilt, egal, welche Werte die Programmvariablen vor Ausführung der Anweisungen hatten.

•

Zusicherungen in einem Programm können dynamisch, d.h. zur Laufzeit, oder statisch, d.h. zur Compilezeit, geprüft werden. Die statische Prüfung von Zusicherungen ist Gegenstand der formalen Programmverifikation. Im Folgenden sollen kurz statische und dynamische Prüfung von Zusicherungen gegenüber gestellt werden und gezeigt werden, daß beide Ansätze Vor- und Nachteile haben. Die dynamische Prüfung von Zusicherungen ist z.B. in Ada und C möglich. Für Ada gibt es das Tool Anna (annotated Ada) [L90], mit dem es unter anderem möglich ist, Zusicherungen zu formulieren und dynamisch zu überprüfen.

Beispiel:

```
x := a;
y := x + 1;
--| y = a+1;
```

Hier ist $y = a + 1$ die Zusicherung. Anna übersetzt ein annotiertes Programm in ein Ada-Programm, in dem die Annotationen durch *checking functions* ersetzt werden. Falls die Zusicherung gilt, wird die nächste Anweisung ausgeführt. Falls die Zusicherung an dieser Stelle nicht gilt, wird eine exception ausgelöst.

•

Zusicherungen, die dynamisch geprüft werden, kann man in C-Programmen mit dem assert-Makro darstellen [C90: 7.2]: **`assert`**`( expression )`

Beispiel:

```
assert(1 <= i && i <= n); y = a[i];
```

Hier ist $1 \le i \le n$ eine Zusicherung. Falls die Zusicherung an dieser Stelle gilt, wird die nächste Anweisung ausgeführt. Falls die Zusicherung an dieser Stelle nicht gilt, wird eine imple-

mentierungsabhängige Fehlermeldung auf die Standardausgabe geschrieben, z.B. `assertion failed: 1 <= i && i <= n, file name.c, line 543,` und die **`abort`**-Funktion aufgerufen.

•

Der Vorteil des assert-Makros besteht darin, daß es zum Standard von C gehört. Anna ist ein Zusatz zu Ada. Installation und Bedienung sind relativ kompliziert. Der Vorteil von Anna besteht in der Mächtigkeit der Ausdrücke in den Zusicherungen. Als `expression` im assert-Makro von C sind nur C-expressions erlaubt. Anna erlaubt die Verwendung selbst definierter Prädikate in Zusicherungen. Die Prädikate werden über sogenannten *virtuellen Programmtext* definiert. Darüber hinaus kann man quantifizierte Ausdrücke verwenden.
Laufzeitchecks haben gegenüber einer statischen Analyse den *Vorteil*, daß die Zusicherungen immer ausgewertet und damit entschieden werden können. Dazu werden einfach die aktuellen Werte der Variablen in die Zusicherungen eingesetzt. Die in der statischen Analyse auftretenden VCs sind mathematische Aussagen, die bewiesen werden müssen. Da Beweise aus verschiedenen Gründen nicht immer gelingen, kann es vorkommen, daß ein Korrektheitsbeweis nicht geführt werden kann. Das kommt besonders dann vor, wenn die zu beweisende Aussage zu komplex ist.

Beispiel:

```
-- p ≥ 2 ∧ x, y, z ≥ 1 ∧ p, x, y, z ∈ ℤ
p := p + 1;
-- x**p + y**p ≠ z**p
```

Die VC lautet $(\forall p, x, y, z: p \geq 2 \wedge x, y, z \geq 1 \Rightarrow x^{p+1} + y^{p+1} \neq z^{p+1})$. Das ist genau Fermats Theorem, das erst vor Kurzem (relativ zur Dauer, in der Problem nicht gelöst war) bewiesen wurde [CR96]. Vermutlich ist zur Zeit kein automatischer Beweiser in der Lage, diesen Satz zu beweisen, so daß die Korrektheit dieses Programms nicht automatisch nachgewiesen werden kann.

•

Der entscheidende *Nachteil* von Laufzeitchecks ist, daß die Zusicherungen nur für den aktuellen Zustand ausgewertet werden, d.h. man erhält keinen Korrektheitsbeweis, der sicherstellt, daß das Programm für alle möglichen Anfangswerte der Spezifikation entsprechend arbeitet. Das ist dem Testen von Programmen ähnlich. Zudem hat die Verletzung einer Zusicherung zur Laufzeit einen Programmabbruch zur Folge, der natürlich durch geeignete Fehlermeldungen oder exception handler soweit wie möglich abgemildert werden sollte. Laufzeitchecks finden Fehler zur Laufzeit, Programmbeweise finden Fehler zur Compilezeit. Ein weiterer Nachteil, der aber im Allgemeinen nicht so schwer wiegend ist, ist die geringere Geschwindigkeit und der erhöhte Speicherbedarf von Programmen mit Laufzeitchecks.
Es gibt Firmen, die Zusicherungen zur Erhöhung der Zuverlässigkeit ihrer Programme verwenden, z.B. Microsoft [M95] in seinen C- bzw. C++-Programmen. Praxis Critical Systems verwendet SPARK, eine Teilmenge von Ada mit Zusicherungen. Das STARS-Projekt der USAF verwendet auch eine Teilmenge von Ada mit Zusicherungen. Darin geschriebene Programme werden mit dem Tool Penelope, das einen interaktiven Beweiser benutzt, verifiziert [O94/1], [O94/2], [O94/3], [O94/4]. Eine weitere Teilmenge von Ada zur Unterstützung der Programmverifikation ist AVA (a verifiable Ada) [SA95], [S95].

2 Notationen und Begriffe

Die Notation von Formeln, Beweisen usw. in der vorliegenden Arbeit lehnt sich zu einem großen Teil an [G90] und [DS90: 21-29] an, die auch von anderen Autoren übernommen wurde, z.B. [B93] und [F93]. Der Hauptaspekt dieser Notation ist die gut lesbare, rein formale und nicht verbale Präsentation von Beweisen. Dabei hat sich herausgestellt, daß durch die Form der Beweisführung ein großer Beitrag zum Verstehen des Beweises geleistet werden kann. Die Beweisführung sieht im Allgemeinen so aus:

Sei A der zu beweisende Satz.

A

$\equiv$ Hinweis, warum $A \equiv B$

B

$\Leftarrow$ Hinweis, warum $C \Rightarrow B$

C

$\equiv$ Hinweis, warum C gilt

true

Die Hinweise sind arithmetische oder logische Gesetze, wie z.B. De Morgan oder Distributivität, spezielle Eigenschaften von *wp*, wie z.B. Konjunktivität, Verweise auf Sätze oder Lemmata oder einfach nur allgemeine Hinweise, wie z.B. „Logik“. Dadurch kann es zwar vorkommen, daß einige Beweise länger werden, aber die Zeit zum Verstehen eines Beweises hängt nicht von der Länge der schriftlichen Darstellung ab, sondern von der Zeit, die der Leser von einem zum nächsten Schritt braucht. So kann man einen Beweis in vielen kleinen Schritten oft schneller verstehen, als einen Beweis mit wenigen großen.
Oft gelten Sätze nur unter bestimmten Voraussetzungen. Wenn es nicht von vornherein klar ist, warum eine Voraussetzung notwendig ist, wird die Notwendigkeit der Voraussetzung durch ein Beispiel gezeigt. Auch wenn Sätze eine Implikation und keine Äquivalenz sind, wird, wenn es notwendig erscheint, durch ein Beispiel gezeigt, warum der Satz nur in einer Richtung gilt. Dieses Vorgehen ist zwar im rein formalen Sinne nicht notwendig, erleichtert aber oft das Verständnis. Das Ende jeder abgeschlossenen Einheit (oder Block im Jargon der Programmiersprachen) wie z.B. eines Beispiels oder Satzes wird durch ein • gekennzeichnet.
Für verschiedene Kategorien von Text werden verschiedene Schriftarten verwendet. Neben dem normalen Text in dieser Schriftart gibt es `Programmtext in dieser Schriftart` (wobei **`reservierte Worte fett`** sind) und Formeln, wie z.B. $(\forall i{:}1 \le i \le n-1{:}\, b(i) \le b(i+1))$.
In Definitionen wird der zu definierende Begriff *kursiv* geschrieben oder der zu definierende Begriff wird von seiner Definition durch $:=$ bzw. $:\equiv$ getrennt. Programmtext besteht aus Anweisungen in Ada-Syntax und Zusicherungen in Form von Ada-Kommentaren. Mit Programm ist eine Sequenz von Anweisungen gemeint, nicht ein vollständiges Programm im Sinne von Ada. In den Zusicherungen sind mathematische Notationen zulässig, auch solche, die nicht der Ada-Syntax genügen.

Beispiel:

```
-- x ≥ 3y ∧ (∃ j: 1 ≤ j < 2n: prim(j))
if x >= y then y := 2 * n; end if;
```

•

Für die Äquivalenz von Prädikaten und die semantische Äquivalenz von Programmen wird das Zeichen ≡ verwendet, sonst das übliche Zeichen =, z.B. für arithmetische Ausdrücke und Mengen. Die folgende Liste enthält die in der Arbeit verwendeten Operatoren nach Priorität

geordnet (Operator niedrigster Priorität in der obersten Zeile, Operatoren mit gleicher Priorität in einer Zeile):

Logische Operatoren	$\Rightarrow, \Leftarrow$		
	$\equiv$		
	$\vee$		
	$\wedge$		
	$\neg$		
Relationale Operatoren	$<, \leq, >, \geq, =, \neq$	Mengenoperatoren	$\subseteq, \subset, \supseteq, \supset$
Arithmetische Operatoren	$+, -$		$\cup$
	$\cdot, /$		$\cap, \setminus$
	einstelliges -		$\overline{\ }$ (Komplement)

Wenn es notwendig erscheint, werden für eine bessere Lesbarkeit Teilausdrücke explizit geklammert, auch wenn die Klammerung wegen der Priorität der Operatoren überflüssig ist.

•

Zu jedem binären und assoziativen Operator o gibt es einen Quantor O [Co90: 45-59]. Quantifizierte Ausdrücke werden so dargestellt: $(\mathrm{O}\ v{:}\, P(v){:}\, E(v))$. Dabei ist v eine Liste von Variablen, P ein Prädikat über diesen Variablen und E ein Ausdruck über v, dessen Typ von O abhängt. P wird im Allgemeinen dazu verwendet, eine Einschränkung oder einen Bereich der betrachteten Variablen anzugeben. $P \equiv \mathit{true}$ bedeutet keine Einschränkung, $P \equiv \mathit{false}$ bedeutet Quantifizierung über den leeren Bereich. In diesem Fall ist der quantifizierte Ausdruck gleich bzw. äquivalent dem neutralen Element n(o) des zugrunde liegenden Operators o, also $(\mathrm{O}\, v{:}\, \mathit{false}{:}\, E(v)) = n(\mathrm{o})$. Zu folgenden Operatoren werden Quantoren verwendet:

Operator	Quantor	Alternative Schreibweise	Neutrales Element n(o)	Typ von E
+	Σ	Sum	0	Arithmetisch
•	Π	Prod	1	Arithmetisch
Min	Min		maxint	Arithmetisch
Max	Max		minint	Arithmetisch
$\wedge$	$\forall$	Forall	True	Logisch
$\vee$	$\exists$	Exists	False	Logisch
$\cap$	$\cap$	Intersection	$\overline{\varnothing}$	Menge
$\cup$	$\cup$	Union	$\varnothing$	Menge

Da in realen Programmen alle ganzen Zahlenbereiche endlich sind, gibt es immer größte und kleinste ganze Zahlen, *maxint* und *minint*.

Außerdem gilt $(\forall v{:}\, P(v){:}\, E(v)) \equiv (\forall v{:}\, P(v) \Rightarrow E(v))$ und $(\exists v{:}\, P(v){:}\, E(v)) \equiv (\exists v{:}\, P(v) \wedge E(v))$.

Lemma 2.1 (O-split) [Co90: 47]:

$$(\{i \mid P_1(i)\} \cup \{i \mid P_2(i)\} = \{i \mid P(i)\}) \wedge (\{i \mid P_1(i)\} \cap \{i \mid P_2(i)\}) = \varnothing \Rightarrow$$
$$(\mathrm{O}\ v{:}\, P(v){:}\, E(v)) = (\mathrm{O}\, v{:}\, P_1(v){:}\, E(v))\ \mathrm{o}\ (\mathrm{O}\, v{:}\, P_2(v){:}\, E(v)).$$

•

Bemerkung: Für die idempotenten Operatoren $\wedge$, $\vee$, $\cap$, $\cup$, min und max ist die 2. Voraussetzung des Lemmas (die Bereiche müssen disjunkt sein) nicht notwendig.

2.1 Grundlegende Definitionen und Sätze

Im Folgenden ist *v* eine Variable, *e* ein beliebiger Ausdruck, *S* ein Programm sowie *P* und *R* prädikatenlogische Ausdrücke. Alle in *P* frei vorkommenden Variablen werden durch den everywhere-Operator [*P*] [DS90: 8-10] allquantifiziert.

Definition 2.2: *Free*(*e*) ist die Menge der freien Variablen im Ausdruck *e* und *Var*(*S*) die Menge der Programmvariablen.
Bemerkung: auf eine formale, induktive Definition von *frei* wird hier verzichtet, siehe z.B. [G81: 76-77].

•

Definition 2.3 (Transparenz) [BMW89]: Sei *S* ein Programm und $v \in Var(S)$. Dann bedeutet $trans(S,v)$: *S* ist *transparent* für *v*, d.h. *v* wird von *S* nicht verändert.

•

Bemerkung: Für Ada bedeutet das, daß *v* nicht auf der linken Seite einer Wertzuweisung oder als out- oder in out-Parameter verwendet werden darf. Damit sind auch Anweisungsfolgen wie `v := v + 1; v := v - 1` ausgeschlossen. Außerdem darf *v* nicht durch Seiteneffekte verändert werden. Somit ist die Transparenz eine rein syntaktische bzw. textuelle Eigenschaft. Daraus ergibt sich
Lemma 2.4: $trans(S1;S2,v) \equiv trans(S1,v) \wedge trans(S2,v)$.

•

Die folgenden 4 Substitutionsregeln sind für den Beweiser sehr wichtig. Durch sie ist es möglich, die Anzahl der Variablen in einem zu beweisenden Satz oder einem Teil davon zu verringern, möglicherweise auf Kosten eines komplizierteren Ausdrucks. Aber im Allgemeinen hängt der Aufwand eines automatischen Beweises zum größten Teil von der Anzahl der Variablen ab.

Lemma 2.5 (Substitution) [DS90: 119]: Es gilt $[v = e \wedge P \equiv v = e \wedge P_e^v]$.

Lemma 2.6 (One-point rule) [DS90: 66]: Für $v \notin Free(e)$ gilt $[(\forall v: v = e: P) \equiv P_e^v]$.

Lemma 2.7 (One-point rule) [BMW89]: Für $v \notin Free(e)$ gilt $[(\exists v: v = e: P) \equiv P_e^v]$.

Lemma 2.8 (Substitution): Sei $v \notin Var(e)$. Dann gilt $[(\forall v: P \wedge v = e \Rightarrow R) \equiv (P_e^v \Rightarrow R_e^v)]$.
Beweis:

$(\forall v : P \wedge v = e \Rightarrow R)$

$\equiv$ shuffle (Lemma 2.10)

$(\forall v : v = e \Rightarrow \neg P \vee R)$

$\equiv$

$(\forall v : v = e : \neg P \vee R)$

$\equiv$ One-point rule

$(\neg P \vee R)_e^v$

$\equiv$ Lemma 2.9

$\neg P_e^v \vee R_e^v$

$\equiv$ Logik

$(P_e^v \Rightarrow R_e^v)$

•

Lemma 2.9 [DS90: 116]: Für einen Operator *f* und Ausdrücke $e_1,\ldots,e_n,e$ gilt:

$$f(e_1,\ldots,e_n)_e^v \equiv f(e_{1_e}^{\ v},\ldots,e_{n_e}^{\ v}).$$

•

Bemerkung: In mathematischen Aussagen mit freien Variablen spielt die Reihenfolge der Operationen *Substitution von Variablen durch Ausdrücke* und *Anwendung mathematischer / logischer Gesetze* keine Rolle, da alle Variablen mit gleichem Namen den gleichen Wert repräsentieren (referentielle Transparenz). Dies gilt jedoch nicht für die Funktionen *wp* und *LP*, die später definiert werden, da es in Formeln, die *wp* bzw. *LP* enthalten, vorkommen kann, daß ein Variablenname unterschiedliche Werte repräsentiert. Im nächsten Kapitel wird gezeigt, unter welchen Voraussetzungen dieses Lemma auch für die Funktion *wp* gilt (Lemmata 3.18 und 3.19 (Substitutionslemma für *wp*)).

•

Es folgen einige häufig vorkommende Transformationen, für die es in der Literatur keine weit verbreiteten Namen gibt, wie z.B. Kommutativität oder De Morgan.

Lemma 2.10 (shuffle) [Co90: 39]: $(a \Rightarrow b \vee c) \equiv (a \wedge \neg c \Rightarrow b)$.

Lemma 2.11 (portation) [H93: 230]: $(a \Rightarrow (b \Rightarrow c)) \equiv (a \wedge b \Rightarrow c)$.

Lemma 2.12 (Substitutionslemma): Seien *P* ein Prädikat, *x* und *y* verschiedene Variablen und *e* und *f* Ausdrücke. Dann gilt $[x = e \wedge y = f \Rightarrow (P_e^x)_f^y \equiv P_{e_f^y,f}^{x,y}]$.

Beweis:

$[x = e \wedge y = f \Rightarrow (P_e^x)_f^y \equiv P_{e_f^y,f}^{x,y}]$

≡ Logik

$[x = e \wedge y = f \Rightarrow ((P_e^x)_f^y \Rightarrow P_{e_f^y,f}^{x,y}) \wedge (P_{e_f^y,f}^{x,y} \Rightarrow (P_e^x)_f^y)]$

≡ Logik

$[x = e \wedge y = f \Rightarrow ((P_e^x)_f^y \Rightarrow P_{e_f^y,f}^{x,y})] \wedge [x = e \wedge y = f \Rightarrow (P_{e_f^y,f}^{x,y} \Rightarrow (P_e^x)_f^y)]$

≡ Portation

$[x = e \wedge y = f \wedge (P_e^x)_f^y \Rightarrow P_{e_f^y,f}^{x,y}] \wedge [x = e \wedge y = f \wedge P_{e_f^y,f}^{x,y} \Rightarrow (P_e^x)_f^y]$

≡ mehrfache Substitution

$[x = e \wedge y = f \wedge P \Rightarrow x = e_f^y \wedge y = f \wedge P] \wedge [x = e \wedge y = f \wedge x = e_f^y \wedge P \Rightarrow x = e \wedge y = f \wedge P]$

≡ Logik

$[x = e \wedge y = f \wedge P \Rightarrow x = e_f^y] \wedge true$

≡ Substitution

$[(x = e)_f^y \wedge y = f \wedge P_f^y \Rightarrow x = e_f^y]$

≡ Lemma 2.9

$[x_f^y = e_f^y \wedge y = f \wedge P_f^y \Rightarrow x = e_f^y]$

≡ *x* und *y* verschieden

$[x = e_f^y \wedge y = f \wedge P_f^y \Rightarrow x = e_f^y]$

≡ Logik

true

•

Bemerkung:

1. Lemma 4.4.8 aus [G81: 81] ist ein Spezialfall davon.
2. Die Bedingung, daß *x* und *y* verschiedene Variablen sein müssen, ist schon aus rein syntaktischen Gründen notwendig. Denn wie soll in einem Ausdruck die gleiche Variable *gleichzeitig* durch 2 möglicherweise verschiedene Ausdrücke ersetzt werden?

•

Lemma 2.13 (case analysis):

$$[(\exists i{:}1 \le i \le n{:}b_i)] \Rightarrow ([a \Rightarrow (\exists i{:}1 \le i \le n{:}b_i \wedge c_i)] \Leftarrow [(\forall i{:}1 \le i \le n{:}a \wedge b_i \Rightarrow c_i)]) .$$

Beweis:

$$[(\exists i : 1 \le i \le n : b_i)] \Rightarrow ([a \Rightarrow (\exists i : 1 \le i \le n : b_i \wedge c_i)] \Leftarrow [(\forall i : 1 \le i \le n : a \wedge b_i \Rightarrow c_i)])$$

$$\equiv 2 \times \text{portation}$$

$$[(\exists i : 1 \le i \le n : b_i)] \wedge [(\forall i : 1 \le i \le n : a \wedge b_i \Rightarrow c_i) \wedge a] \Rightarrow [(\exists i : 1 \le i \le n : b_i \wedge c_i)]$$

$$\Leftarrow \text{Logik}$$

$$[(\exists i : 1 \le i \le n : b_i) \wedge (\forall i : 1 \le i \le n : a \wedge b_i \Rightarrow c_i) \wedge a \Rightarrow (\exists i : 1 \le i \le n : b_i \wedge c_i)]$$

$$\equiv \text{gebundene Umbenennung, Skolemisierung}$$

$$[(\forall j : 1 \le j \le n \wedge b_j \wedge (\forall i : 1 \le i \le n : a \wedge b_i \Rightarrow c_i) \wedge a \Rightarrow (\exists i : 1 \le i \le n : b_i \wedge c_i))]$$

$$\equiv \text{mit } i = j$$

$$[(\forall j : 1 \le j \le n \wedge b_j \wedge (\forall i : 1 \le i \le n : a \wedge b_i \Rightarrow c_i) \wedge a \Rightarrow 1 \le j \le n \wedge b_j \wedge c_j)]$$

$$\equiv \text{Logik, modus ponens}$$

$$[(\forall j : 1 \le j \le n \wedge b_j \wedge (\forall i : 1 \le i \le n : b_i \Rightarrow c_i) \wedge a \Rightarrow c_j)]$$

$$\Leftarrow \text{Logik}$$

$$[(\forall j : 1 \le j \le n \wedge b_j \wedge c_j \Rightarrow c_j)]$$

$$\equiv \text{Logik}$$

true

•

Bemerkung: Diese Transformation ist nur anwendbar, wenn einer der Fälle b_i, nach denen die Fallunterscheidung durchgeführt wird, gilt. Formeln dieser Form treten bei der Berechnung der schwächsten Vorbedingung des Case-Statements auf. In Ada ist die obige Voraussetzung immer erfüllt. Es folgt je ein Beispiel für die Notwendigkeit der Voraussetzung und die Nicht-Äquivalenz.

•

Beispiel (Notwendigkeit der Voraussetzung):
Die Voraussetzung wird von $[x > 0 \vee x < 0]$ nicht erfüllt. Daher gilt auch nicht

$$[x = a \Rightarrow x > 0 \wedge x = |a| \vee x < 0 \wedge -x = |a|]$$
$$\Leftarrow$$
$$[(x = a \wedge x > 0 \Rightarrow x = |a|) \wedge (x = a \wedge x < 0 \Rightarrow -x = |a|)]$$

für $x = a = 0$.

•

Beispiel (Nicht-Äquivalenz):
Die Voraussetzung wird durch $[x \geq 0 \vee x \leq 0 \vee x = 1]$ erfüllt, aber die beiden Prädikate

$$[x = a \Rightarrow x \geq 0 \wedge x = |a| \vee x \leq 0 \wedge -x = |a| \vee x = 1 \wedge -x = a]$$

und

$$[(x = a \wedge x \geq 0 \Rightarrow x = |a|) \wedge (x = a \wedge x \leq 0 \Rightarrow -x = |a|) \wedge (x = a \wedge x = 1 \Rightarrow -x = a)]$$

sind nicht äquivalent, z.B. für $x = a = 1$

•

Ein häufiger Spezialfall der letzten Regel ist das If-Statement. Dabei ist $n = 2$ mit $b_2 \equiv \neg b_1$. Dann gilt sogar Äquivalenz.
Lemma 2.14 (case analysis): $(a \Rightarrow b \wedge c \vee \neg b \wedge d) \equiv ((a \wedge b \Rightarrow c) \wedge (a \wedge \neg b \Rightarrow d))$.
Beweis:

$(a \wedge b \Rightarrow c) \wedge (a \wedge \neg b \Rightarrow d)$

$\equiv$ Logik

$(\neg a \vee \neg b \vee c) \wedge (\neg a \vee b \vee d)$

$\equiv$ Logik

$\neg a \vee \neg a \wedge b \vee \neg a \wedge d \vee \neg a \wedge \neg b \vee \neg b \wedge d \vee \neg a \wedge c \vee b \wedge c \vee d \wedge c$

$\equiv$ Logik

$\neg a \vee \neg b \wedge d \vee b \wedge c \vee d \wedge c$

$\equiv$ Logik

$a \Rightarrow \neg b \wedge d \vee b \wedge c \vee d \wedge c$

$\equiv \vee$-Subsumierung $(d \wedge c \Rightarrow \neg b \wedge d \vee b \wedge c)$

$a \Rightarrow \neg b \wedge d \vee b \wedge c$

•

Diese Transformationsregeln werden so angewendet, daß die linke Seite durch die rechte Seite ersetzt wird. Die Ersetzungen sind in *mexana* implementiert und sind Teil der Transformation einer Formel in eine Konjunktion von Klauseln.

2.2 Relationale Semantik

Zwei wesentliche Aspekte von Programmiersprachen sind Syntax und Semantik. Die Syntax beschreibt das Aussehen oder den Aufbau eines Programmes in der Programmiersprache. Die Semantik beschreibt die Bedeutung oder die Wirkung des Programmes, d.h. was das Programm macht. Die Syntax einer Programmiersprache wird im Allgemeinen durch eine kontextfreie Grammatik formal beschrieben [TS87: 38], [TD95: 479-502]. Für die formale Beschreibung der Semantik gibt es im Wesentlichen 3 Möglichkeiten [W93]:

- Operationelle Semantik
- Denotationelle Semantik
- Relationale Semantik oder axiomatische Semantik

Die operationelle Semantik beschreibt die Wirkung eines Programmes durch einen semantisch äquivalenten Ausdruck in einer anderen formalen Sprache, also durch die Operationen in einer anderen Sprache, diurch die dieses Programm ausgeführt werden kann. Das kann eine

andere konkrete Programmiersprache sein oder die Sprache einer abstrakten Maschine wie eine Zwischensprache. Diese Methode der Semantikbeschreibung wird z.B. bei der Codegenerierung im Compilerbau verwendet oder zur Beschreibung der Wirkung von bestimmten Konstrukten einer Programmiersprache [ISO90: 6.8.3.9].
Die denotationale Semantik beschreibt die Wirkung eines Programmes oder Ausdrucks durch eine partielle Semantikfunktion [W93: 56]. Die Semantikfunktion *A* für Ausdrücke bildet von der Menge der Ausdrücke *Exp* und Zustände *Z* in die Menge *W* der Werte des entsprechenden Ausdrucks ab, $A: Exp \times Z \to W$. Die Semantikfunktion *P* für Programme bildet von der Menge der Anweisungen *C* und Zustände *Z* in die Menge der Folgezustände *Z* ab. Diese Methode der Semantikbeschreibung findet z.B. bei funktionalen Programmiersprachen Anwendung.

2.2.1 Programmzustände

Wie überzeuge ich mich selbst oder jemand anderen, daß ein von mir geschriebenes Programm auch das tut, was es soll? Zunächst muß genau festgelegt werden, was das Programm tun soll, d.h. die Semantik oder Wirkung des Programms muß genau festgelegt werden. Das ist die Programmspezifikation. Sie ist im Allgemeinen verbal bzw. in natürlicher Sprache abgefaßt. Da natürliche Sprache im Allgemeinen nicht eindeutig bzw. mißverständlich ist, ist sie für die formale Spezifikation nicht gut geeignet. Aus diesem Grund wird eine formale Beschreibung der Semantik benötigt. Hat man eine formale Spezifikation, kann man mit formalen, mathematischen Mitteln prüfen, ob das gegebene Programm die Spezifikation erfüllt. Wenn ja, nennt man das Programm korrekt. Die verbale Aussage „das Programm ist korrekt“ wird dabei in eine mathematische Aussage transformiert, die zu beweisen versucht wird. Der Beweis kann, wie alle mathematischen Beweise, von einem Menschen, einer Maschine oder einer Kombination von beiden durchgeführt werden. Dabei bestehen natürlich alle bekannten Schwierigkeiten des Beweisens. Für eine genauere Beschreibung der relationalen Semantik werden zunächst einige Definitionen benötigt.

Definition 2.15 (Zustand): $V=\{v_1,..., v_n\}$, $n \geq 1$, sei die Menge der Programmvariablen, im Folgenden Variablen. $w_1,..., w_n$ seien die entsprechenden Wertebereiche, die durch die Typen der Variablen gegeben sind, wobei die w_i nicht leer und endlich sind. Ein *Programmzustand*, im Folgenden *Zustand*, ist eine Funktion *s* von der Menge der Programmvariablen *V* in die Menge ihrer möglichen Werte

$$W = \bigcup_{i=1}^{n} w_i, \quad s: V \to W; \quad s(v_i) \in w_i.$$

Es wird vereinbart, daß der Zustand einer Variablen erst nach einer expliziten Zuweisung definiert ist.

•

Beispiel:

```
subtype t1 is natural range -10 .. 10;
type t2 is array(1 .. 10) of t1;
x1: boolean;
x2: t1 := 0;
a: t2 := (-2, 0, 3, -1, 5, -6, 1, 0, 10, 1);
```

$s(x1)$ ist undefiniert,

$s(x2) = 0,\ s(a) = \{(1,-2),(2,0),(3,3),(4,-1),(5,5),(6,-6),(7,1),(8,0),(9,10),(10,1)\}$.

Bemerkung: *a* ist ein Array, mathematisch eine Funktion, weshalb der Wert von *a* auch eine Funktion ist, deren Wert durch die Menge ihrer Paare angegeben wird.

Wenn man die *n* Variablen total ordnet, was immer möglich ist, kann man einen Zustand durch Angabe eines *n*-Tupels beschreiben.

Definition 2.16 (Zustandsmenge): Die Menge aller möglichen Zustände wird als *Zustandsmenge Z* bezeichnet. Es gilt

$$|Z| = \prod_{i=1}^{|V|} |w_i| \geq 1.$$

•

Der Tupel-Schreibweise für einen Zustand entspricht eine Menge von Tupeln für eine Zustandsmenge.

Beispiel: (Angabe der Zustandsmenge als Menge von Paaren)

```
subtype t is natural range 0 .. 2;
x: t := readint;
b: boolean := readint < 0;
-- ZM =
{(0,  false),(0,  true),(1,  false),(1,  true),(2,  false),(2,
true)}
```

•

Definition 2.17 (Charakteristisches Prädikat) [B95: 33]: Sei *Z* sie Menge aller möglichen Zustände und $X \subseteq Z$.
Dann heißt das durch $(\forall s \in Z: C(X)(s) \equiv s \in X)$ beschriebene Prädikat $C(X)$ das *charakteristische Prädikat* der Zustandsmenge *X*.
Dabei ist für ein Prädikat *P* und einen Zustand *s* $P(s) :\equiv P^{v1\cdots vn}_{s(v1)\cdots s(vn)}$.

•

Beispiel:

```
subtype t is natural range 0 .. 2;
x1, x2: t := readint;
if x1 < x2
-- ZM = {(0,1),(0,2),(1,2)}; C(ZM) ≡ 0 ≤ x1 < x2 ≤ 2
then ...
```

•

Lemma 2.18 [B95: 33]: Sei *PS* die Menge aller Äquivalenzklassen von Prädikaten.
Die Funktion C: $2^Z \rightarrow PS$ ist bijektiv.

•

Satz 2.19 (Äquivalenz von Zustandsmengen und Prädikaten) [B95: 33]:
Die beiden vollständigen Verbände $(2^Z, \subseteq, \cap, \cup, \bar{}, \varnothing, Z)$ und $(PS, \Rightarrow, \wedge, \vee, \neg, \mathit{false}, \mathit{true})$ sind über die Funktion *C* isomorph.

•

Auf Grund dieser Äquivalenz spielt es theoretisch keine Rolle, ob man Zustandsmengen oder ihre charakteristischen Prädikate verwendet. Für die praktische Verwendung, d.h. die Anwendung auf konkrete Programme, in Zusammenhang mit einem automatischen oder auch menschlichen Beweiser ist die Verwendung von Prädikaten jedoch vorteilhafter. Für abstrakte Betrachtungen der Theorie ist die Verwendung von Zustandsmengen vorteilhafter, siehe z.B. Satz 2.31 und Abschnitt 2.3. Grob gesprochen kann man sagen, daß für Aussagen *über wp* die Verwendung von Zustandsmengen vorteilhafter ist und für Aussagen *mit wp* die Verwendung der charakteristischen Prädikate, wie es auch in [B95: 78] formuliert ist.

Lemma 2.20: Da *C* bijektiv ist, gibt es eine Umkehrfunktion C^{-1}, die jedem Prädikat *P* eine Zustandsmenge $X \subseteq Z$ zuordnet, so daß gilt $C^{-1}(P) = X \equiv (\forall s \in Z: s \in X \equiv P(s))$.

•

Definition 2.21 (Zusicherung): Zustandsmengen beschreibende Prädikate in einem Programm heißen *Zusicherung*. Ein Programm mit Zusicherungen heißt *annotiert*. Die Zusicherung unmittelbar vor einer Anweisung heißt *Vorbedingung* (*precondition*), die Zusicherung nach einer Anweisung heißt *Nachbedingung* (*postcondition*).

•

Die *relationale Semantik* [B95: 56] beschreibt die Wirkung eines Programms durch den Zustand vor Ausführung (Vorbedingung) und den Zustand nach Ausführung (Nachbedingung) des Programms, d.h. durch die durch das Programm bewirkte *Zustandsänderung*. Die Zustandsänderung eines Programms wird auf die Zustandsänderungen der einfachen Anweisungen zurückgeführt. Die durch die einfachen Anweisungen bewirkten Zustandsänderungen definieren die Semantik der einfachen Anweisungen. Die Wirkung eines Programms wird dann auf Grund von Deduktionsregeln [G81] hergeleitet. Diese relationale Semantik wird bei der formalen Programmverifikation und damit in der vorliegenden Arbeit verwendet.

Beispiel:

```
x, y, m: integer := readint;
-- x, y, m ∈ integer
if x>y then
   -- x, y, m ∈ integer ∧ x > y
   m := x;
   -- x, y, m ∈ integer ∧ m = x ∧ x > y
else
   -- x, y, m ∈ integer ∧ x ≤ y
   m := y;
   -- x, y, m ∈ integer ∧ m = y ∧ x ≤ y
end if;
-- x, y, m ∈ integer ∧ m ≥ x ∧ m ≥ y ∧ (m = x ∨ m = y) ≡
-- x, y, m ∈ integer ∧ m = max (x, y);
```

•

2.2.2 Korrektheit

Nach der formalen Definition der Semantik ist es möglich, die Korrektheit eines Programms und die semantische Äquivalenz formal zu definieren. Ein Programm ist dann korrekt, wenn es tut, was es tun soll. Was es tun soll, ist die *Spezifikation* des Programms. In der relationalen Semantik besteht eine Spezifikation aus einer Vor- und einer Nachbedingung. Man kann also nur von Korrektheit eines Programms bezüglich einer Spezifikation sprechen. Das Programm ist dann korrekt, wenn es in einem Zustand startet, der die Vorbedingung erfüllt und in einem Zustand endet, der die Nachbedingung erfüllt. (Hier wird implizit davon ausgegangen, daß zur Korrektheit auch die Terminierung gehört, das nennt man totale Korrektheit, im Gegensatz zur partiellen Korrektheit, womit gemeint ist: falls das Programm terminiert, dann in einem Zustand, der die Nachbedingung erfüllt [DS90: 129]).
Dazu ermittelt man die größte Menge *LP* (*largest preset*) [W96] von Anfangszuständen, die garantieren, daß nach Start des Programms in einem dieser Anfangszustände das Programm in einem Zustand endet, der die Nachbedingung erfüllt. Für die Definition von *LP* benötigt man noch die *Programmfunktion* f_P eines Programms *P* [W96], [MM96]. (In [MM96] wird ähnlich wie in [W96] ein relationales Modell eines Programms verwendet, jedoch mit einem anderen Ziel, das für die vorliegende Arbeit nicht von Bedeutung ist). Diese Funktion ordnet

jedem Anfangszustand diejenige Menge von Endzuständen zu, in denen sich das Programm nach Terminierung befinden kann. Nichtdeterministische Programme können nach Start in einem bestimmten Anfangszustand in verschiedenen Zuständen enden, daher ist der Wert von f_P eine Zustandsmenge. Deterministische Programme enden nach Start in einem bestimmten Anfangszustand immer im gleichen Zustand. Nicht alle Programme, die in einem definierten Anfangszustand gestartet werden, terminieren. Nicht alle Programme, die in einem definierten Anfangszustand gestartet werden, erreichen einen definierten Endzustand (z.B. bei Division durch 0). Daher könnte man zunächst f_P als partielle Funktion definieren, die nur auf den Zuständen definiert ist, die das Programm in einem definierten Zustand terminieren lassen. Dieser Ansatz hat aber 2 Nachteile:

1. Ein Programm kann in jedem definierten Zustand gestartet werden; man kann nicht von vornherein ausschließen, daß es in einem Zustand gestartet wird, der zu einer Endlosschleife oder einem undefinierten Ausdruck führt. So ist z.B. in der Mathematik die Division definiert als Funktion $\backslash : \mathbb{R} \times \mathbb{R} \setminus \{0\} \mapsto \mathbb{R}$. Trotzdem kann man auf jedem normalen Taschenrechner 5 / 0 eingeben. Das Ergebnis ist dann so etwas wie ERROR.
2. Dieser Punkt greift auf die *LP*- bzw. *wp*-Funktion zurück, die formal erst durch die Definitionen 2.24 und 2.26 eingeführt werden. Wenn man ein nicht terminierendes Programm oder ein in einem undefinierten Zustand terminierendes Programm dadurch charakterisieren würde, daß das Ergebnis der Programmfunktion die leere Menge ist, hätte man $wp(P, true) \equiv true$ unabhängig davon, ob P terminiert oder nicht. (Beweis: $wp(P, true)$ entspricht $LP(P, Z)$. $LP(P, Z) := \{s \in Z: f_P(s) \subseteq Z\}$. Nach Definition wäre immer $f_P(s) \subseteq Z$. Bei nicht terminierenden Programmen wäre $f_P(s) = \varnothing$ und die Teilmengenbeziehung wäre auch erfüllt. Damit wäre $LP(P, Z) = \{s: s \in Z\} = Z$.). Auch das *law of the excluded miracle* (Satz 2.38) würde dann nicht gelten. Denn dann gäbe es Anfangszustände, die zu keinem definierten Endzustand führen würden, z.B. solche, die zu einer Endlosschleife führen. Siehe auch die Bemerkung zu Satz 2.38.

Diese beiden Nachteile werden dadurch behoben, daß man f_P als totale Funktion in die Menge der Zustände, erweitert um einen undefinierten Zustand $\perp$, definiert. Die Menge der Endzustände eines nicht terminierenden oder in einem undefinierten Zustand terminierenden Programms enthält dann $\perp$.

Definition 2.22 (Programmfunktion) [W86]: Der *undefinierte Zustand* heißt $\perp$, $\perp \notin Z$, $Z_\perp := Z \cup \{\perp\}$. Die Funktion $f_P: Z \mapsto 2^{Z_\perp}$ ordnet jedem Zustand $s \in Z$ die Menge aller Zustände $f_P(s)$ zu, in denen sich das Programm *P* nach Start in *s* befinden kann. Da f_P total ist, gilt $(\forall s \in Z: f_P(s) \neq \varnothing)$. Programme mit der Eigenschaft $(\forall s \in Z: |f_P(s)| = 1)$ heißen *deterministisch*. Für nicht terminierende oder in einem undefinierten Zustand terminierende Programme gilt $(\exists s \in Z: \perp \in f_P(s))$.

•

Bemerkung zum letzten Satz der Definition: Der Existenzquantor wird verwendet, weil ein Programm schon dann als nicht terminierend oder in einem undefinierten Zustand terminierend bezeichnet wird, wenn es einen Zustand mit dieser Wirkung gibt.
Nichtdeterministische Programme können vom selben Anfangszustand aus in einem definierten Zustand enden oder in einem undefinierten oder in einer Endlosschleife. Schon wenn so eine Möglichkeit besteht, wird es als nicht terminierend oder in einem undefinierten Zustand terminierend bezeichnet. Daher wird in der Definition $\perp \in f_P(s)$ und nicht $f_P(s) = \{\perp\}$ verwendet. Zu implizitem Nichtdeterminismus siehe auch die Bemerkung zu Satz 2.35.

Es wird nicht zwischen nicht terminierend und in einem undefinierten Zustand terminierend unterschieden. Falls diese Unterscheidung gebraucht wird, kann sie leicht eingefügt werden.

•

Definition 2.23 (Normal terminierend): Ein Programm P heißt *normal terminierend*, wenn $(\forall s \in Z: \perp \notin f_P(s))$ gilt.

•

Definition 2.24 (largest preset) [W86]: Die Menge aller Anfangszustände *s*, die nach Ausführung des Programms *P* in einen Zustand aus $N \subseteq Z_\perp$ führen, heißt *largest preset* $LP(P,N)$ des Programms *P* und des Endzustandes *N*. $LP(P,N) := \{s \in Z: f_P(s) \subseteq N\}$.

•

Satz 2.25 [W86]: *LP*(*P*,*N*) ist die größte Menge mit der eben definierten Eigenschaft.

•

Definition 2.26 (weakest precondition): Das charakteristische Prädikat *C*(*LP*) des *largest preset* eines Programms *S* und einer Nachbedingung *N* heißt *weakest precondition* von *S* und *N*, $wp(S,N) :\equiv C(LP(S, C^{-1}(N)))$.

•

Definition 2.27 (Korrektheit): Ein Programm *S* ist bezüglich einer Spezifikation (*V*, *N*), $(V,N) \in 2^Z \times 2^Z$, genau dann *korrekt*, wenn $V \subseteq LP(S,N)$.

•

Bemerkung: Jedes korrekte Programm soll in einem definierten Zustand terminieren, d.h. für eine Nachbedingung *N* gilt nie $\perp \in N$. Daher $(V,N) \in 2^Z \times 2^Z$ und nicht $(V,N) \in 2^Z \times 2^{Z_\perp}$.

•

Satz 2.28 (Korrektheit): Ein Programm *S* ist bezüglich einer Spezifikation (*V*, *N*), $V, N \in Rp(PS)$, genau dann korrekt, wenn $[V \Rightarrow wp(S,N)]$. (*Rp*(*PS*) ist eine Menge von Repräsentanten von *PS*; damit sind *V* und *N* Prädikate.)

Beweis:

$\Rightarrow$:

Ein Programm *S* ist bezüglich einer Spezifikation (*V*, *N*), $V, N \in Rp(PS)$, korrekt

$\equiv$ Äquivalenz von Zustandsmengen und Prädikaten

Ein Programm *S* ist bezüglich einer Spezifikation $((C^{-1}(V), C^{-1}(N)), C^{-1}(V), C^{-1}(N) \in 2^Z$, korrekt

$\equiv$ Definition Korrektheit und $x \in 2^M \equiv x \subseteq M$

$C^{-1}(V) \subseteq LP(S, C^{-1}(N))$

$\Rightarrow$ Äquivalenz von Zustandsmengen und Prädikaten

$[C(C^{-1}(V)) \Rightarrow C(LP(S, C^{-1}(N)))]$

$\equiv$ Lemma 2.20, Definition *wp*

$[V \Rightarrow wp(S,N)]$

$\Leftarrow$:

$[V \Rightarrow wp(S,N)]$

$\equiv$ Logik

$[V \wedge wp(S,N) \equiv V]$

$\equiv$ Definition *wp*

$[V \wedge C(LP(S,C^{-1}(N))) \equiv V]$

$\Rightarrow$ Leibniz

$C^{-1}(V \wedge C(LP(S,C^{-1}(N)))) = C^{-1}(V)$

$\equiv$ Isomorphie

$C^{-1}(V) \cap C^{-1}(C(LP(S,C^{-1}(N)))) = C^{-1}(V)$

$\equiv$ Lemma 2.20

$C^{-1}(V) \cap LP(S,C^{-1}(N)) = C^{-1}(V)$

$\equiv$ Mengenlehre

$C^{-1}(V) \subseteq LP(S,C^{-1}(N))$

$\equiv$ Definition Korrektheit und $x \in 2^M \equiv x \subseteq M$

Ein Programm *S* ist bezüglich einer Spezifikation $((C^{-1}(V),C^{-1}(N)),C^{-1}(V),C^{-1}(N) \subseteq 2^Z$, korrekt

$\equiv$ Äquivalenz von Zustandsmengen und Prädikaten

Ein Programm *S* ist bezüglich einer Spezifikation (*V*, *N*), $V, N \in Rp(PS)$, korrekt

2.2.3 Semantische Äquivalenz

2 Programme *S*1 und *S*2 sind genau dann *semantisch äquivalent* $S1 \equiv S2$, wenn sie dieselbe Wirkung haben. Diese intuitive Definition kann man nun mit Hilfe der relationalen Semantik formalisieren. Wenn eine Anweisung eine beliebige Spezifikation (*V*, *N*) erfüllt, dann auch die andere Anweisung und umgekehrt. Da es hier um beliebige Spezifikationen geht, müssen *V* und *N* allquantifiziert werden.

Bemerkung:

```
-- true              -- true
x := 1;  und         x := 2;
-- x>0               -- x>0
```

gelten. Beide Anweisungen erfüllen also die eine gleiche Spezifikation, haben aber offensichtlich nicht die gleiche Wirkung. Es reicht also nicht aus, daß es eine Spezifikation gibt, die von beiden Anweisungen erfüllt wird.

•

Damit hat man

Definition 2.29: Zwei Anweisungen *S*1 und *S*2 sind genau dann *semantisch äquivalent* $S1 \equiv S2$, wenn

$$\begin{bmatrix} --V & & --V \\ S1 & \equiv & S2 \\ --N & & --N \end{bmatrix}.$$

•

Der folgende Satz sagt aus, daß die semantische Äquivalenz gleichbedeutend mit der Äquivalenz der schwächsten Vorbedingungen ist. Damit wird der Nachweis der semantischen Äquivalenz zweier Programme erleichtert. Der Grund dafür, warum die semantische Äquivalenz nicht gleich durch den folgenden Satz definiert wird, ist der, daß Definition 2.29 dem intuitiven Verständnis von semantischer Äquivalenz näher kommt. Für den Beweis des Satzes werden Zustandsmengen statt ihrer charakteristischen Prädikate verwendet. Zuvor benötigt man

Lemma 2.30: $(\forall B, D : (\forall A : A \subseteq B \Rightarrow A \subseteq D) \equiv B \subseteq D)$.

Satz 2.31: Die semantische Äquivalenz ist äquivalent zu

$$[(\forall N : wp(S1,N) \equiv wp(S2,N)]) .$$

Beweis: zu zeigen ist

$$[(\forall V, N : (V \Rightarrow wp(S1,N)) \equiv (V \Rightarrow wp(S2,N)))] \equiv [(\forall N : wp(S1,N) \equiv wp(S2,N))] .$$

$[(\forall V, N : (V \Rightarrow wp(S1,N)) \equiv (V \Rightarrow wp(S2,N)))] \equiv [(\forall N : wp(S1,N) \equiv wp(S2,N))]$

$\equiv$ Definition *wp*

$[(\forall V, N : C^{-1}(V) \subseteq LP(S1, C^{-1}(N)) \equiv C^{-1}(V) \subseteq LP(S2, C^{-1}(N)))] \equiv$

$[(\forall N : LP(S1, C^{-1}(N)) = LP(S2, C^{-1}(N)))]$

$\equiv$ mit $A := C^{-1}(V), B := LP(S1, C^{-1}(N)), D := LP(S2, C^{-1}(N))$

$[(\forall A, B, D : (A \subseteq B \equiv A \subseteq D) \equiv B = D)]$

$\equiv$ Logik

$[(\forall A, B, D : (A \subseteq B \Rightarrow A \subseteq D) \wedge (A \subseteq D \Rightarrow A \subseteq B) \equiv B = D)]$

$\equiv$ Lemma 2.30

$[(\forall B, D : B \subseteq D \wedge D \subseteq B \equiv B = D)]$

$\equiv$ Mengenlehre

true

•

Um also zu zeigen, daß zwei Anweisungen die gleiche Wirkung haben, muß man zeigen, daß ihre schwächsten Vorbedingungen für alle Nachbedingungen äquivalent sind. Mit dieser formalen Definition kann man intuitive Behauptungen wie „das sieht man ja, daß das gleich ist“ beweisen oder widerlegen. Außerdem hat man ein Mittel, solche Beweise zu automatisieren, denn was sich formalisieren läßt, läßt sich auch automatisieren, zumindest im Prinzip.

2.3 Eigenschaften von *wp*

Für das weitere Vorgehen werden einige Eigenschaften der Funktion *wp* benötigt. Diese sind in [G81] informell (mehr oder weniger durch intuitives Einsehen) und in [DS90] rein formal mit Hilfe der Prädikatenlogik bewiesen. Diese Beweise sind allerdings relativ langwierig. In diesem Abschnitt werden diese grundlegenden Eigenschaften mit Hilfe der auf Zustandsmengen operierenden Funktion *LP* gezeigt, wobei sich herausstellt, daß die Beweise wesentlich kürzer und verständlicher werden [W96], [B95: 78]. Im Folgenden werden alle Aussagen über *wp* in äquivalente Aussagen über *LP* transformiert, analog zum Beweis von Satz 2.28, ohne daß diese Transformation jedesmal explizit durchgeführt wird. Die Aussagen werden mit *wp* formuliert, weil sie in dieser Form später benutzt werden.

Satz 2.32 (Konjunktivität) [W96]: $[wp(P, N \wedge M) \equiv wp(P, N) \wedge wp(P, M)]$.

Beweis: zu zeigen ist $LP(P, N \cap M) = LP(P, N) \cap LP(P, M)$.

$LP(P, N \cap M)$

= Definition LP

$\{s \in Z: f_P(s) \subseteq N \cap M\}$

= Mengenlehre

$\{s \in Z: f_P(s) \subseteq N \wedge f_P(s) \subseteq M\}$

= Mengenlehre

$\{s \in Z: f_P(s) \subseteq N\} \cap \{s \in Z: f_P(s) \subseteq M\}$

= Definition LP

$LP(P, N) \cap LP(P, M)$

•

Bemerkung: der Vorteil dieser Beweisführung liegt darin, daß N, M und $f_P(s)$ Mengen sind. Dadurch kann man elementare Mengenlehre anwenden, ohne sich Gedanken über die Bedeutung der Mengen oder Programmabläufe machen zu müssen. Der Beweis ist kurz, einfach und formal.

•

Satz 2.33 (Monotonie) [W96]: $[N \Rightarrow M] \Rightarrow [wp(P, N) \Rightarrow wp(P, M)]$.

Beweis: zu zeigen ist $N \subseteq M \Rightarrow LP(P, N) \subseteq LP(P, M)$.

$N \subseteq M$

≡ Mengenlehre

$N \cap M = N$

⇒ Leibniz

$LP(P, N \cap M) = LP(P, N)$

≡ Konjunktivität

$LP(P, N) \cap LP(P, M) = LP(P, N)$

≡ Mengenlehre

$LP(P, N) \subseteq LP(P, M)$

•

Bemerkung: In [Co90: 85] ist dieser Satz folgendermaßen dargestellt [W96/1]:

$$(N \Rightarrow M) \Rightarrow (wp(P, N) \Rightarrow wp(P, M)).$$

Da hier Prädikate mit freien Variablen verwendet werden, und nach Konvention alle freien Variablen allquantifiziert sind, ist der Satz mit dem everywhere-Operator so zu lesen: $[(N \Rightarrow M) \Rightarrow (wp(P, N) \Rightarrow wp(P, M))]$. Dann ist er aber nicht mehr gültig, z.B. für $N \equiv true, M \equiv x = 1, P \equiv x := x + 1$ ergibt sich $[x = 1 \Rightarrow x = 0]$. An diesem Beispiel sieht man einen weiteren Vorteil in der Verwendung von Zustandsmengen. Sie enthalten keine freien Variablen, und bei der Rücktransformation in Prädikate gibt es keine Missverständnisse über die Platzierung des everywhere-Operators bzw. der Klammerung.

•

Satz 2.34 (schwache Disjunktivität) [W96]: $[wp(P, N \vee M) \Leftarrow wp(P, N) \vee wp(P, M)]$.

Beweis: zu zeigen ist $LP(P, N \cup M) \supseteq LP(P, N) \cup LP(P, M)$.

$LP(P, N \cup M)$

= Definition LP

$\{s \in Z: f_P(s) \subseteq N \cup M\}$

$\supseteq$ Mengenlehre

$\{s \in Z: f_P(s) \subseteq N \vee f_P(s) \subseteq M\}$

= Mengenlehre

$\{s \in Z: f_P(s) \subseteq N\} \cup \{s \in Z: f_P(s) \subseteq M\}$

= Definition LP

$LP(P, N) \cup LP(P, M)$

•

Bemerkung: Disjunktivität ist nicht analog zu Konjunktivität. Die Disjunktivität ist nur eine Implikation und keine Äquivalenz. Auf diesen Unterschied kommt man auch hier rein formal mit elementarer Mengenlehre: $(\forall \textit{Mengen } A, B, C : A \subseteq B \cup C \Leftarrow A \subseteq B \vee A \subseteq C)$.
Äquivalenz gilt nur, wenn $|A| \leq 1$ oder $B \subseteq C \vee C \subseteq B$.

•

Satz 2.35 (Starke Disjunktivität) [W96]: Für deterministische Programme P gilt

$[wp(P, N \vee M) \equiv wp(P, N) \vee wp(P, M)]$.

Beweis: zu zeigen ist $LP(P, N \cup M) = LP(P, N) \cup LP(P, M)$.

$LP(P, N \cup M)$

= Definition LP

$\{s \in Z: f_P(s) \subseteq N \cup M\}$

= Mengenlehre, $|f_P(s)| = 1$

$\{s \in Z: f_P(s) \subseteq N \vee f_P(s) \subseteq M\}$

= Mengenlehre

$\{s \in Z: f_P(s) \subseteq N\} \cup \{s \in Z: f_P(s) \subseteq M\}$

= Definition LP

$LP(P, N) \cup LP(P, M)$

•

Bemerkung: Zunächst folgt ein Beispiel für die Nicht-Äquivalenz der schwachen Disjunktivität bei nicht deterministischen Programmen.

$wp(x := 1 \lozenge x := 2, x = 1 \vee x = 2) \equiv \textit{true}$, aber

$wp(x := 1 \lozenge x := 2, x = 1) \vee wp(x := 1 \lozenge x := 2, x = 2) \equiv \textit{false} \vee \textit{false} \equiv \textit{false}$ (Kein Anfangszustand garantiert, daß nach Ausführung der nichtdeterministischen Zuweisung $x = 1$ gilt. Andererseits gilt nach Ausführung der nichtdeterministischen Zuweisung $x = 1 \vee x = 2$.).
Auch Programmiersprachen, die scheinbar deterministisch sind, beinhalten im Allgemeinen impliziten Nichtdeterminismus. Z.B. ist in Ada und anderen imperativen Sprachen die Rei-

henfolge der Auswertung von Ausdrücken oder aktuellen Parametern nicht festgelegt. Ein Beispiel für diesen impliziten Nichtdeterminismus ist

```
function f(p: integer) return integer is
begin
  z := z + 1;
  return p + 1;
end if;
-- x = y = z = 0
x := f(y) + z;
```

Der Anfangszustand kann abhängig von der Reihenfolge der Auswertung des Ausdrucks `f(y) + z` zu den beiden Endzuständen $x = 1$ bzw. $x = 2$ führen. Der Grund dafür ist hier die globale Variable *z*. Man darf also die starke Disjunktivität nur unter der Voraussetzung deterministischer Programme verwenden, insbesondere muß auch impliziter Nichtdeterminismus ausgeschlossen sein.

Definition 2.36: Zwei Prädikate *P* und *Q* heißen *vergleichbar*, wenn [P⇒Q] oder [Q⇒P] gilt.

Lemma 2.37 (Starke Disjunktivität): Für vergleichbare Prädikate *N* und *M* gilt

$$[wp(P, N \vee M) \equiv wp(P, N) \vee wp(P, M)].$$

Beweis: O.B.d.A. gelte nach der Voraussetzung $[N \Rightarrow M]$.

$[N \Rightarrow M]$
≡ Logik
$[N \vee M \equiv M]$
⇒ Leibniz
$[wp(P, N \vee M) \equiv wp(P, M)]$

und

$[N \Rightarrow M]$
⇒ Monotonie
$[wp(P, N) \Rightarrow wp(P, M)]$
≡ Logik
$[wp(P, N) \vee wp(P, M) \equiv wp(P, M)]$

•

Bemerkung: Das läßt sich auf jede endliche Kette von Prädikaten verallgemeinern.

•

Satz 2.38 (law of the excluded miracle) [G81: 110]: $[wp(P, \mathit{false}) \equiv \mathit{false}]$.

Beweis: zu zeigen ist $LP(P, \varnothing) = \varnothing$.

$LP(P, \varnothing)$
= Definition *LP*
$\{s \in Z: f_P(s) \subseteq \varnothing\}$
= Mengenlehre
$\{s \in Z: f_P(s) = \varnothing\}$
= Definition Programmfunktion, $(\forall s \in Z: f_P(s) \neq \varnothing)$

$\{s \in Z: \mathit{false}\}$

= Mengenlehre

$\varnothing$

•

Bemerkung: Es gibt also keinen Anfangszustand, der zu keinem Endzustand führt, oder anders ausgedrückt, alle Anfangszustände führen zu einem Endzustand. Das gilt also auch für nicht terminierende oder in einem undefinierten Zustand terminierende Programme. Dann ist der Endzustand $\perp$. Ohne die Erweiterung der Zustandsmenge um $\perp$ wäre dieser Satz so nicht gültig.

•

Satz 2.39 (Terminierung): Ein Programm P terminiert genau dann normal, wenn

$$[wp(P, \mathit{true}) \equiv \mathit{true}]\,.$$

Beweis:

$LP(P, Z) = Z$

≡ Definition LP

$\{s \in Z: f_P(s) \subseteq Z\} = Z$

≡ Mengenlehre

$\{s \in Z: f_P(s) \subseteq Z\} \subseteq Z \wedge \{s \in Z: f_P(s) \subseteq Z\} \supseteq Z$

≡ Mengenlehre

$\mathit{true} \wedge \{s \in Z: f_P(s) \subseteq Z\} \supseteq Z$

zu zeigen bleibt also (mit der Definition für *normal terminierend*) $(\forall s \in Z: \perp \notin f_P(s)) \equiv Z \subseteq \{s \in Z: f_P(s) \subseteq Z\}$.

⇐:

$Z \subseteq \{s \in Z: f_P(s) \subseteq Z\} \Rightarrow (\forall s \in Z: \perp \notin f_P(s))$

≡ Mengenlehre

$(\forall x \in Z: x \in \{s \in Z: f_P(s) \subseteq Z\}) \Rightarrow (\forall s \in Z: \perp \notin f_P(s))$

≡ Mengenlehre

$(\forall x \in Z: x \in Z \wedge f_P(x) \subseteq Z) \Rightarrow (\forall s \in Z: \perp \notin f_P(s))$

≡ Kontraposition

$(\exists s \in Z: \perp \in f_P(s)) \Rightarrow (\exists x \in Z: f_P(x) \not\subset Z)$

≡ Skolemisierung

$(\forall s \in Z: \perp \in f_P(s) \Rightarrow (\exists x \in Z: f_P(x) \not\subset Z))$

≡ mit $x = s$

$(\forall s \in Z: \perp \in f_P(s) \Rightarrow s \in Z \wedge f_P(s) \not\subset Z)$

≡ Logik, Mengenlehre

$(\forall s \in Z: \perp \in f_P(s) \Rightarrow (\exists x \in f_P(s): x \notin Z))$

$\equiv$ mit $x = \bot$

$(\forall s \in Z: \bot \in f_P(s) \Rightarrow \bot \in f_P(s) \wedge \bot \notin Z)$

$\equiv$ Logik, Definition Programmfunktion

true

$\Rightarrow$:

$(\forall s \in Z: \bot \notin f_P(s)) \Rightarrow Z \subseteq \{s \in Z: f_P(s) \subseteq Z\}$

$\equiv$ Mengenlehre

$(\forall s \in Z: \bot \notin f_P(s)) \Rightarrow (\forall x: x \in Z \Rightarrow x \in \{s \in Z: f_P(s) \subseteq Z\})$

$\equiv$ Mengenlehre

$(\forall s \in Z: \bot \notin f_P(s)) \Rightarrow (\forall x: x \in Z \Rightarrow x \in Z \wedge f_P(x) \subseteq Z)$

$\equiv$ Skolemisierung

$(\forall x: (\forall s \in Z: \bot \notin f_P(s)) \Rightarrow (x \in Z \Rightarrow x \in Z \wedge f_P(x) \subseteq Z))$

$\equiv$ portation

$(\forall x: (\forall s \in Z: \bot \notin f_P(s)) \wedge x \in Z \Rightarrow x \in Z \wedge f_P(x) \subseteq Z)$

$\equiv$ Logik

$(\forall x: (\forall s \in Z: \bot \notin f_P(s)) \wedge x \in Z \Rightarrow f_P(x) \subseteq Z)$

$\Leftarrow$ modus ponens

$(\forall x: \bot \notin f_P(x) \wedge x \in Z \Rightarrow f_P(x) \subseteq Z)$

$\equiv$ Logik, Definition *Z*, Definition Programmfunktion

$(\forall x \in Z: \bot \notin Z \wedge f_P(x) \subseteq Z_\bot \wedge \bot \notin f_P(x) \Rightarrow f_P(x) \subseteq Z)$

$\equiv$ 2 mal shuffle

$(\forall x \in Z: \bot \notin Z \wedge f_P(x) \subset Z \wedge \bot \notin f_P(x) \Rightarrow f_P(x) \subset Z_\bot)$

$\equiv$ Mengenlehre

$(\forall x \in Z: \bot \notin Z \wedge (\exists y: y \in f_P(x) \wedge y \notin Z) \wedge \bot \notin f_P(x) \Rightarrow (\exists s: s \in f_P(x) \wedge s \notin Z_\bot))$

$\equiv$ Skolemisierung, Definition Programmfunktion

$(\forall x \in Z, y: \bot \notin Z \wedge y \in f_P(x) \wedge y \notin Z \wedge \bot \notin f_P(x) \Rightarrow (\exists s: s \in f_P(x) \wedge s \notin Z \cup \{\bot\}))$

$\equiv$ Mengenlehre

$(\forall x \in Z, y: \bot \notin Z \wedge y \in f_P(x) \wedge y \notin Z \wedge \bot \notin f_P(x) \Rightarrow (\exists s: s \in f_P(x) \wedge s \notin Z \wedge s \neq \bot))$

$\equiv \forall$-split

$(\forall x \in Z, y: y = \bot \wedge \bot \notin Z \wedge y \in f_P(x) \wedge y \notin Z \wedge \bot \notin f_P(x) \Rightarrow (\exists s: s \in f_P(x) \wedge s \notin Z \wedge s \neq \bot)) \wedge$
$(\forall x \in Z, y: y \neq \bot \wedge \bot \notin Z \wedge y \in f_P(x) \wedge y \notin Z \wedge \bot \notin f_P(x) \Rightarrow (\exists s: s \in f_P(x) \wedge s \notin Z \wedge s \neq \bot))$

$\equiv$ Substitution

$(\forall x \in Z : \bot \notin Z \wedge \bot \in f_P(x) \wedge \bot \notin Z \wedge \bot \notin f_P(x) \Rightarrow (\exists s : s \in f_P(x) \wedge s \notin Z \wedge s \neq \bot)) \wedge$
$(\forall x \in Z, y : y \neq \bot \wedge \bot \notin Z \wedge y \in f_P(x) \wedge y \notin Z \wedge \bot \notin f_P(x) \Rightarrow (\exists s : s \in f_P(x) \wedge s \notin Z \wedge s \neq \bot))$

≡ Logik

$(\forall x \in Z : \mathit{false} \Rightarrow (\exists s : s \in f_P(x) \wedge s \notin Z \wedge s \neq \bot)) \wedge$
$(\forall x \in Z, y : y \neq \bot \wedge \bot \notin Z \wedge y \in f_P(x) \wedge y \notin Z \wedge \bot \notin f_P(x) \Rightarrow (\exists s : s \in f_P(x) \wedge s \notin Z \wedge s \neq \bot))$

≡ Logik

$(\forall x \in Z, y : y \neq \bot \wedge \bot \notin Z \wedge y \in f_P(x) \wedge y \notin Z \wedge \bot \notin f_P(x) \Rightarrow (\exists s : s \in f_P(x) \wedge s \notin Z \wedge s \neq \bot))$

≡ mit $s = y$

$(\forall x \in Z, y : y \neq \bot \wedge \bot \notin Z \wedge y \in f_P(x) \wedge y \notin Z \wedge \bot \notin f_P(x) \Rightarrow y \in f_P(x) \wedge y \notin Z \wedge y \neq \bot)$

≡ Logik
true

•

Bemerkung: Wenn man sichergestellt hat, daß P normal terminiert, kann man also $wp(P, true)$ durch *true* ersetzen.
Für nicht normal terminierende Programme gilt nicht $[wp(P, true) \equiv true]$.
Beweis durch Widerspruch: angenommen, es gilt $[wp(P, true) \equiv true]$, also

$LP(P, Z) = Z$

≡ Definition LP

$\{s \in Z : f_P(s) \subseteq Z\} = Z$

≡ Voraussetzung: P nicht normal terminierend

$\{s \in Z : \{\bot\} \subseteq Z\} = Z$

≡ Definition Z

$\{s \in Z : \mathit{false}\} = Z$

≡ Logik, Mengenlehre

$\varnothing = Z$

≡ Definition Z

false

•

Lemma 2.40 (Konsequenzregel):

$$[A \Rightarrow B] \wedge [C \Rightarrow D] \wedge [B \Rightarrow wp(P, C)] \Rightarrow [A \Rightarrow wp(P, D)].$$

Beweis: zu zeigen ist $M \subseteq N \wedge R \subseteq S \wedge N \subseteq LP(P, R) \Rightarrow M \subseteq LP(P, S)$

$M \subseteq N \wedge R \subseteq S \wedge N \subseteq LP(P, R) \Rightarrow M \subseteq LP(P, S)$

⇐ Transitivität von $\subseteq$

$R \subseteq S \wedge M \subseteq LP(P,R) \Rightarrow M \subseteq LP(P,S)$

$\equiv$ Definition LP

$R \subseteq S \wedge M \subseteq \{s \in Z: f_P(s) \subseteq R\} \Rightarrow M \subseteq \{s \in Z: f_P(s) \subseteq S\}$

$\equiv$ Mengenlehre

$(\forall x \in M: R \subseteq S \wedge x \in Z \wedge f_P(x) \subseteq R) \Rightarrow M \subseteq \{s \in Z: f_P(s) \subseteq S\}$

$\Leftarrow$ Transitivität von $\subseteq$

$(\forall x \in M: x \in Z \wedge f_P(x) \subseteq S) \Rightarrow M \subseteq \{s \in Z: f_P(s) \subseteq S\}$

$\equiv$ Mengenlehre

$M \subseteq \{s \in Z: f_P(s) \subseteq S\} \Rightarrow M \subseteq \{s \in Z: f_P(s) \subseteq S\}$

$\equiv$ Logik

true

3 Verifikationsbedingungen

3.1 Überblick und Begriffsbestimmung

In diesem Kapitel werden für verschiedene Anweisungen Bedingungen, die die Korrektheit implizieren (*Verifikationsbedingungen VCs*), dargestellt. Dabei sind die meisten Ergebnisse nicht neu, sondern die Art und Weise, wie sie erzielt werden. Die Semantik der Anweisungen P wird nicht durch $wp(P, N)$ definiert, sondern durch ihre Programmfunktion $f_P(s)$. $wp(P, N)$ wird dann als Satz formuliert. Die Umsetzung der verbalen Semantikbeschreibung einer Anweisung P in $wp(P, N)$ ist nicht so klar und intuitiv einleuchtend wie die Umsetzung der verbalen Semantikbeschreibung einer Anweisung P in $f_P(s)$. Der tiefere Grund dafür ist wahrscheinlich, daß die verbalen Semantikbeschreibungen, wie sie in den meisten Programmiersprachen vorherrschen, vorwärts gerichtet sind. Sie sagen, was gemacht wird, und zwar in der zeitlichen Reihenfolge. Die Funktion *wp* ist eher rückwärts gerichtet und daher nicht so anschaulich. Das wird in den folgenden Abschnitten deutlich. In [B95: 75-76] werden, ausgehend von einer verbalen, informellen Semantikbeschreibung, zwei Funktionen *wp* und $\overline{wp}$ definiert, die eine Formalisierung der informellen Semantikbeschreibung darstellen. Dann wird gezeigt, daß diese beiden Definitionen äquivalent sind [B95: 80]. Hier wird, ausgehend von einer verbalen, informellen Semantikbeschreibung, die Semantik einer Anweisung durch die Definition der Programmfunktion formalisiert. Die *wp*-Funktion wird als Satz formuliert, wobei ihr Beweis die Programmfunktion benutzt.
Zudem wird die explizite Typprüfung bei der Zuweisung eingeführt (Abschnitt 3.2.2), das Substitutionslemma für *wp* formuliert (Abschnitt 3) und eine neue VC für For-Schleifen vorgestellt (Abschnitt 3.4.2). Die Korrektheit von Schleifen wird durch eine Konjunktion von 5 (bei While-Schleifen, Abschnitt 3.4.1) bzw. 4 Bedingungen (bei For-Schleifen, Abschnitt 3.4.2), in denen eine Invariante und Terminierungsfunktion (bei While-Schleifen) vorkommen, impliziert. Es wird formal gezeigt, daß auch die Umkehrung dieser Implikation (bis auf eine in der Praxis unbedeutende Ausnahme) gilt (Abschnitte 3.4.1 und 3.4.2). Anhand von 5 Beispielen wird gezeigt, daß zum Beweis der Korrektheit tatsächlich alle 5 Bedingungen notwendig sind. Es folgt ein Vergleich von 4 in der Literatur vorkommenden Vorbedingungen für Prozeduraufrufe (Abschnitt 3.4.3.2). Außerdem werden leistungsfähige VCs für Schleifen im Kontext einer umgebenden Anweisung (Abschnitt 3.4.4) und rekursive Prozeduren vorgestellt (Abschnitt 3.4.5). Eine Schleife im Kontext einer umgebenden Anweisung ist z.B. eine Schleife, die im Then-Teil eines If-Statement vorkommt (dann ist die umgebenden Anweisung das If-Statement) oder eine innere Schleife bei geschachtelten Schleifen (dann ist die umgebenden Anweisung die äußere Schleife).

Nach Satz 2.28 ist ein Programm P bezüglich einer Spezifikation (V, N) genau dann korrekt, wenn $[V \Rightarrow wp(P, N)]$ gilt. Für den Korrektheitsnachweis sind im Wesentlichen also 2 Schritte erforderlich:

1. Berechnung des Prädikates $wp(P, N)$
2. Überprüfung, ob die Implikation gilt

Die 2 Schritte müssen in der angegebenen Reihenfolge durchgeführt werden. Der 2. Schritt kann vollständig oder teilweise von einem automatischen Beweiser durchgeführt werden und ist nicht Gegenstand dieser Arbeit. Die Berechnung des Prädikates $wp(P, N)$ ist abhängig von P nicht immer möglich. Daher ist ein allgemeineres Vorgehen die Verwendung einer *Verifikationsbedingung VC*. Das ist ein Prädikat, das die Korrektheit impliziert.

Definition 3.1: Für ein Programm *P* und eine Spezifikation (*V*, *N*) ist eine *exakte Verifikationsbedingung VC* das Prädikat $[V \Rightarrow wp(P, N)]$.

•

Da $wp(P, N)$ nicht immer berechnet werden kann, wie z.B. für Schleifen und Prozeduraufrufe, muß man in diesen Fällen eine andere VC benutzen, die stärker als $[V \Rightarrow wp(P, N)]$ ist.

Definition 3.2: Ein Prädikat VC, für das $[VC \Rightarrow [V \Rightarrow wp(P, N)]]$ gilt, ist eine *approximierte VC*.

•

Bemerkung: Natürlich ist man daran interessiert, eine möglichst schwache VC zu benutzen, so daß man *false* nicht verwenden wird. Es kann auch sein, daß eine approximierte VC nicht nur die Korrektheit impliziert (was natürlich für jede approximierte VC gezeigt werden muß), sondern sogar zur Korrektheit äquivalent ist. Das kann aber nicht entschieden werden, wenn $wp(P, N)$ nicht berechnet werden kann.

•

Für eine approximierte VC gilt nur VC $\Rightarrow$ Korrektheit und nicht unbedingt die umgekehrte Richtung. Das bedeutet, daß man in so einem Fall ein korrektes Programm möglicherweise nicht beweisen kann. Ein Beispiel dafür befindet sich im Beweis von Satz 3.34, Teil 3 und Beispiel 7 von Abschnitt 3.4.3.2. Es kann aber nicht sein, daß man ein nicht korrektes Programm beweisen kann. Zunächst werden die Programme *P* betrachtet, für die $wp(P, N)$ berechnet werden kann.

3.2 Exakte Verifikationsbedingungen

Für das Null-Statement, die Zuweisung (unter gewissen Voraussetzungen) sowie das If- und Case-Statement und die nichtdeterministische Auswahl gibt es eine exakte VC. Im Folgenden bezeichnet *N* eine Nachbedingung und $M := C^{-1}(N)$.

3.2.1 Null-Statement

Das Null-Statement `null` macht nichts, es wird keine Variable geändert. Die Wirkung von `null` ist also: jeder Zustand *s* bleibt unverändert, also

$$(\forall s \in Z: f_{Null}(s) = \{s\}).$$

Lemma 3.3 (Null-Regel): $wp(Null, N) \equiv N$.
Beweis: zu zeigen ist $LP(Null, M) = M$ für $M \subseteq Z$

$LP(Null, M)$

= Definition *LP*

$\{s \in Z: f_{Null}(s) \subseteq M\}$

= Definition Programmfunktion, Mengenlehre

$\{s \in Z: s \in M\}$

zu zeigen bleibt $M = \{s \in Z: s \in M\}$:

$M = \{s \in Z: s \in M\}$

$\equiv$ Mengenlehre

$(\forall x: x \in M \equiv x \in \{s \in Z: s \in M\})$

≡ Mengenlehre

$(\forall x: x \in M \equiv x \in Z \wedge x \in M)$

≡ Logik

$(\forall x: (x \in M \Rightarrow x \in Z \wedge x \in M) \wedge (x \in Z \wedge x \in M \Rightarrow x \in M))$

≡ Logik

$(\forall x: (x \in M \Rightarrow x \in Z) \wedge true)$

≡ Mengenlehre

$M \subseteq Z$

≡ Voraussetzung

true

•

Bemerkung: Hier wurde die Null-Regel postuliert und ihre Korrektheit ohne verbale Beschreibung rein formal bewiesen. Man kann die Null-Regel auch als Definition des Null-Statements auffassen. Hier wurde das Null-Statement durch seine Programmfunktion definiert und daraus die Null-Regel postuliert und bewiesen. Diese Definition spiegelt die verbale Definition (kein Zustand wird geändert) genauer wider als die Definition durch *wp*. Das wird bei der Zuweisung noch deutlicher.

•

3.2.2 Zuweisung

Die Wertzuweisung v := e setzt die Variable *v* auf den Wert des Ausdrucks *e* (Falls *v* vor der Zuweisung definiert war und den gleichen Wert wie *e* hatte, wird der Wert von *v* nicht geändert. Hier soll aber der allgemeinere Fall, daß *e* vor der Zuweisung einen beliebigen Wert hat, betrachtet werden). Die Wertzuweisung soll keinen Seiteneffekt bewirken, d.h. es soll nur der Wert der Variablen *v* geändert werden und die Werte aller anderen Variablen bleiben unverändert. Falls Seiteneffekte auftreten, ist die folgende Herleitung der Zuweisungsregel und damit die Zuweisungsregel selbst nicht gültig. Weiterhin soll die Auswertung des Ausdrucks *e* normal terminieren, d.h. nicht zu einem undefinierten Zustand oder einer Endlosschleife führen. Das kann z.B. bei Division durch 0, durch einen Index-out-of-range-error bei einem Arrayzugriff, durch einen arithmetischen Überlauf oder Auswertung einer Funktion passieren. Seiteneffekte bei der Wertzuweisung sind eine Quelle des impliziten Nichtdeterminismus. Eine andere Quelle ist eine nicht initialisierte Variable im Ausdruck *e*. Diese kann zu einem undefinierten Zustand oder zu einem definierten Zustand mit beliebigem Wert für *v* führen. Aus diesem Grund sollen alle Variablen in *e* initialisiert sein, d.h. vor Auswertung von *e* ist das Programm in einem definierten Zustand. Unter diesen Voraussetzungen gilt für die Programmfunktion der Zuweisung

$$(\forall s \in Z: f_{v:=e}(s) = \{s_e^v\}) .$$

Dabei ist s_e^v derselbe Zustand wie *s* bis auf die Stelle *v*, an der der Wert von *e* im Zustand *s* angenommen wird, formal:

Definition 3.4: Für einen Zustand *s* ist

$$s_e^v(x) := \begin{cases} s(x), falls x \neq v \\ e^{v1,\ldots,vn}_{s(v1),\ldots s(vn)} \in W_v, falls x = v \end{cases}$$

•

Beispiel: $s = ((x,1),(y,2))$, $v = x$, $e = x + y$. s_e^v ist derselbe Zustand wie *s* bis auf die Stelle *x*. An der Stelle *y* steht also der Wert 2. An der Stelle *x* steht der Wert von $x + y$ im Zustand *s*, also 3. Damit ist $s_{x+y}^x = ((x,3),(y,2))$. Nach Definition 3.4 ist $s_{x+y}^x(y) = s(y)$ und $s_{x+y}^x(x) = (x + y)_{s(x),s(y)}^{x,y} = (x + y)_{1,2}^{x,y} = 3$.

•

Lemma 3.5 (Zuweisungsregel): $wp(v := e, N) \equiv N_e^v$.

Beweis:

$wp(v := e, N)$

≡ Definition *wp*

$C(LP(v := e, C^{-1}(N)))$

≡ Definition *LP*

$C(\{s \in Z: f_{v:=e}(s) \subseteq C^{-1}(N)\})$

≡ Definition $f_{v:=e}$, Mengenlehre

$C(\{s \in Z: s_e^v \in C^{-1}(N)\})$

zu zeigen bleibt $[C(\{s \in Z: s_e^v \in C^{-1}(N)\}) \equiv N_e^v]$.

$[C(\{s \in Z: s_e^v \in C^{-1}(N)\}) \equiv N_e^v]$

≡ Äquivalenz von Prädikaten

$(\forall t \in Z: C(\{s \in Z: s_e^v \in C^{-1}(N)\})(t) \equiv N_e^v(t))$

≡ Definition Charakteristisches Prädikat

$(\forall t \in Z: t \in \{s \in Z: s_e^v \in C^{-1}(N)\} \equiv N_e^v(t))$

≡ Mengenlehre

$(\forall t \in Z: t \in Z \wedge t_e^v \in C^{-1}(N) \equiv N_e^v(t))$

≡ Definition 3.4

$(\forall t \in Z: t \in Z \wedge t_e^v \in C^{-1}(N) \equiv (N_e^v)_{t(v1),\ldots,t(vn)}^{v1,\ldots,vn})$

≡ Mengenlehre

$(\forall t \in Z: \{t\} \subseteq Z \wedge \{t_e^v\} \subseteq C^{-1}(N) \equiv (N_e^v)_{t(v1),\ldots,t(vn)}^{v1,\ldots,vn})$

≡ Isomorphie

$(\forall t \in Z: [C(\{t\}) \Rightarrow C(Z)] \wedge [C(\{t_e^v\}) \Rightarrow C(C^{-1}(N))] \equiv (N_e^v)_{t(v1),\ldots,t(vn)}^{v1,\ldots,vn})$

≡ Isomorphie

$(\forall t \in Z: [C(\{t\}) \Rightarrow true] \wedge [C(\{t_e^v\}) \Rightarrow N] \equiv (N_e^v)_{t(v1),\ldots,t(vn)}^{v1,\ldots,vn})$

≡ Logik

$(\forall t \in Z:[C(\{t_e^v\}) \Rightarrow N] \equiv (N_e^v)^{v1,\ldots,vn}_{t(v1),\ldots,t(vn)})$

≡ Definition Charakteristisches Prädikat

$(\forall t \in Z:[v_1 = t(v_1) \wedge \cdots \wedge v_n = t(v_n) \wedge v = e^{v1,\ldots,vn}_{t(v1),\ldots,t(vn)} \Rightarrow N] \equiv (N_e^v)^{v1,\ldots,vn}_{t(v1),\ldots,t(vn)})$

≡ Substitution

$(\forall t \in Z: N^{v1,\ldots,vn,v}_{t(v1),\ldots,t(vn),e^{v1,\ldots,vn}_{t(v1),\ldots,t(vn)}} \equiv (N_e^v)^{v1,\ldots,vn}_{t(v1),\ldots,t(vn)})$

≡ Substitutionslemma

$(\forall t \in Z: N^{v1,\ldots,vn,v}_{t(v1),\ldots,t(vn),e^{v1,\ldots,vn}_{t(v1),\ldots,t(vn)}} \equiv N^{v1,\ldots,vn,v}_{t(v1),\ldots,t(vn),e^{v1,\ldots,vn}_{t(v1),\ldots,t(vn)}})$

≡ Logik

true

•

Die Zuweisungsregel gilt unter den oben verbal genannten Voraussetzungen. Einige dieser verbalen Voraussetzungen kann man formalisieren und in die Zuweisungsregel aufnehmen.

1. Die Auswertung von *e* soll nicht in einen undefinierten Zustand führen bedeutet, daß der Wert des Ausdrucks zum Typ der Variablen *v* paßt. Ein undefinierter Zustand bei Auswertung eines Integerausdrucks kann bei Überlauf und Division durch 0 entstehen (damit ist auch mod und rem eingeschlossen). Andere partielle ganzzahlige Funktionen werden durch den nächsten Punkt subsumiert.
2. Die Auswertung eines Ausdrucks *e* kann zu einer Endlosschleife führen, falls *e* einen Funktionsaufruf enthält. Daher soll *e* keinen Funktionsaufruf enthalten. Man kann durch Einführung von Hilfsprozeduren und -variablen einen Ausdruck immer funktionsfrei machen (siehe Abschnitt 3.4.3.1). Dadurch kann zwar die Auswertung von *e* nicht zu einer Endlosschleife führen, die Endlosschleife wird aber in die der Funktion entsprechenden Prozedur verlagert.
3. Alle Variablen sollen einen definierten Wert haben. Das kann dadurch erreicht werden, daß jede Variable bei ihrer Definition initialisiert wird (siehe auch Definition 2.15 (Zustand), letzter Satz und anschließendes Beispiel).

Die Punkte 2 und 3 können rein syntaktisch geprüft werden. Punkt 1 kann durch die *Typbedingung* sichergestellt werden.

Definition 3.6 (Typbedingung): Für eine Wertzuweisung `v := e` ist die *Typbedingung* $T_v(e)$ das Prädikat $e \in W_v$.

•

Damit ergibt sich für die Zuweisungsregel: $wp(v := e, N) \equiv N_e^v \wedge T_v(e)$.

Beispiel:

```
x: Int8; -- Int8 = [-128, 127]
   :
x := -x;
-- x ≥ 0
```

$wp(x := -x, x \geq 0)$

≡ Zuweisungsregel

$-x \geq 0 \wedge -128 \leq -x \leq 127 \wedge -x\ def$

≡ Arithmetik

$x \le 0 \wedge 128 \ge x \ge -127$

≡ Arithmetik

$-127 \le x \le 0$

•

Eine Zuweisung `a(i) := e` eines Ausdrucks *e* zu einer Arraykomponente $a(i)$ ändert den Wert der Arrayvariablen *a* nur an der Stelle *i* unter der Voraussetzung, daß die Auswertung des Ausdrucks *i* normal terminiert und keine Seiteneffekte verursacht. Die Zuweisung kann daher auch in der Form $a := a_e^i$ dargestellt werden, wobei die Variable auf der linken Seite und der Ausdruck auf der rechten Seite vom Typ Funktion sind. a_e^i ist dieselbe Funktion wie *a* außer an der Stelle *i*, wo der Wert *e* angenommen wird, entsprechend der Notation $f[s/t]$ von [SL87: 5]. Damit braucht man keine spezielle Regel für diese Zuweisung. Sie ergibt sich aus der Regel für die Zuweisung zu einer einfachen Variablen. Die Anweisung terminiert normal, wenn die Auswertungen der Ausdrücke *i* und *e* normal terminieren, *i* im Indextyp von *a* liegt und *e* im Komponententyp von *a* liegt. Für die Programmfunktion gilt

$$(\forall s \in Z: f_{a(i):=e}(s) = \{s^a_{a_e^i}\}) .$$

Die Zuweisungsregel ist $wp(a(i) := e, N) \equiv N^a_{a_e^i}$. Mit der Typbedingung ergibt sich

$$wp(a(i) := e, N) \equiv N^a_{a_e^i} \wedge T_{Ka}(e) \wedge T_{Ia}(i) .$$

Dabei ist T_{Ka} der Komponententyp von *a* und T_{Ia} der Indextyp von *a*. Die Regel gilt auch für mehrdimensionale Arrays.

Beispiele:

```
a: array(1..10) of Int8;
i, j: Int8;
   :
a(i) := 0;
-- a(j) = 0
```

$wp(a(i) := 0, a(j) = 0)$

≡ Zuweisungsregel

$(a(j) = 0)^a_{a_0^i} \wedge 1 \le i, j \le 10 \wedge -128 \le 0 \le 127$

≡ Arithmetik

$(i = j \wedge 0 = 0 \vee i \ne j \wedge a(j) = 0) \wedge 1 \le i, j \le 10$

≡ Logik

$(i = j \vee a(j) = 0) \wedge 1 \le i, j \le 10$

•

```
subtype Ind2 is natural range 0..4;
a: array(1..10, Ind2) of Int8;
i: Int8;
j: Ind2;
   :
a(i+1,j) := i + j;
-- a(j,0) ≥ i
```

$wp(a(i+1, j) := i + j, a(j,0) \ge i)$

≡ Zuweisungsregel

$$(a(j,0) \geq i)^{a}_{a^{(i+1,j)}_{i+j}} \wedge 1 \leq i+1,\ j \leq 10 \wedge 0 \leq j, 0 \leq 4 \wedge -128 \leq i+j \leq 127$$

≡ Arithmetik, Definition von *i* und *j*

$$(i+1 = j \wedge j = 0 \wedge i+j \geq i \vee (i+1 \neq j \vee j \neq 0) \wedge a(j,0) \geq i) \wedge 0 \leq i \leq 9 \wedge 1 \leq j \leq 4$$

≡ Arithmetik, Logik

$$(i = -1 \wedge j = 0 \vee a(j,0) \geq i) \wedge 0 \leq i \leq 9 \wedge 1 \leq j \leq 4$$

•

Ähnlich wie Zuweisungen zu Arraykomponenten kann man Zuweisungen zu Recordkomponenten `a.r := e` behandeln. Da die Bezeichner der Recordkomponenten statisch sind, ergeben sich jedoch 2 Vereinfachungen:

1. Die Prüfung, ob *r* eine Komponente von *a* ist, wird von jedem Compiler geprüft und muß nicht als Typbedingung formuliert werden.
2. Bei der Berechnung der schwächsten Vorbedingung muß keine Fallunterscheidung gemacht werden.

Zuweisungen zu Recordkomponenten kann man daher wie Zuweisungen zu einfachen Variablen behandeln. Damit sind auch Zuweisungen zu einer Variable mit einem Selektor, d.h. einer Sequenz von Indizes und Recordkomponenten, zulässig.

Beispiel (*a* ist eine Spezifikationsvariable):

```
        subtype Ind is positive range 1..10;
        type st is record
          left, right: Int8;
        end record;
        type vt is array(Ind) of st:
        type rt is record
          v: vt;
          :
        end record;
        type bt is array(Ind, Ind) of rt;
        b: bt;
        i, j: Int8;
          :
b(i, j).v(3).left := 0;
-- 1 ≤ a ≤ 10 ∧ b(1,a).v(j).left >= 0
```

$$wp(b(i,j).v(3).left := 0, 1 \leq a \leq 10 \wedge b(1,a).v(j).left \geq 0)$$

≡ Zuweisungsregel

$$(b(1,a).v(j).left \geq 0)^{b}_{b_0^{(i,j,v,3,left)}} \wedge 1 \leq 1, a, i, j, 3 \leq 10 \wedge -128 \leq 0 \leq 127$$

≡ Arithmetik

$$(i = 1 \wedge j = a \wedge 3 = j \wedge 0 \geq 0 \vee \neg(i = 1 \wedge j = a \wedge 3 = j \wedge 0 \geq 0) \wedge b(1,a).v(j).left \geq 0) \wedge 1 \leq a, i, j \leq 10$$

≡ Logik

$$(i = 1 \wedge j = a = 3 \vee b(1,a).v(j).left \geq 0) \wedge 1 \leq a, i, j \leq 10$$

3.2.3 Sequenz

Die Wirkung der Sequenz $P1;P2$ von 2 Programmen $P1$ und $P2$ wird wieder über ihre Programmfunktion definiert. Alle Endzustände von $P1$ sind Anfangszustände von $P2$. Jeden dieser Zustände bildet $P2$ auf eine Menge von Endzuständen von $P1;P2$ ab. Da die Programmfunktion nur auf Z definiert ist, muß $P1$ normal terminieren. Damit ist

$$\bot \notin f_{P1}(s) \Rightarrow (\forall u \in Z : f_{P1;P2}(u) = \bigcup_{t \in f_{P1}(u)} f_{P2}(t)) \ .$$

Lemma 3.7 (Sequenzregel): Falls $P1$ normal terminiert, gilt

$$[wp(P1;P2,N) \equiv wp(P1, wp(P2,N))] \ .$$

Beweis: zu zeigen ist $LP(P1;P2,M) = LP(P1,LP(P2,M))$

$LP(P1;P2,M) = LP(P1,LP(P2,M))$

≡ Definition LP

$\{s \in Z: f_{P1;P2}(s) \subseteq M\} = \{s \in Z: f_{P1}(s) \subseteq \{t \in Z: f_{P2}(t) \subseteq M\}\}$

≡ Mengenlehre

$(\forall x: x \in \{s \in Z: f_{P1;P2}(s) \subseteq M\} \equiv x \in \{s \in Z: f_{P1}(s) \subseteq \{t \in Z: f_{P2}(t) \subseteq M\}\})$

≡ Mengenlehre

$(\forall x: x \in Z \wedge f_{P1;P2}(x) \subseteq M \equiv x \in Z \wedge f_{P1}(x) \subseteq \{t \in Z: f_{P2}(t) \subseteq M\})$

≡ Definition $f_{P1;P2}$

$(\forall x : x \in Z \wedge \bigcup_{t \in f_{P1}(x)} f_{P2}(t) \subseteq M \equiv x \in Z \wedge f_{P1}(x) \subseteq \{t \in Z : f_{P2}(t) \subseteq M\})$

≡ Mengenlehre

$(\forall x: x \in Z \wedge (\forall t \in f_{P1}(x): f_{P2}(t) \subseteq M) \equiv x \in Z \wedge (\forall y \in f_{P1}(x): y \in Z \wedge f_{P2}(y) \subseteq M))$

≡ $P1$ terminiert normal

true

•

Die Sequenzregel gilt für mehrere Anweisungen $P1;\ldots;Pn$ entsprechend:

$wp(P1;\ldots Pn, N)$

≡ Sequenzregel

$wp(P1;\ldots Pn-1, wp(Pn, N))$

≡ Sequenzregel

:

≡ Sequenzregel

$wp(P1, wp(P2,\ldots, wp(Pn, N)\cdots)$

Beispiel (Vertauschung):

```
x, y: Int8;
   :
x := x + y;
y := x - y;
x := x - y;
-- x = a ∧ y = b
```

$wp(x := x + y; y := x - y; x := x - y, x = a \wedge y = b)$

≡ Sequenzregel

$wp(x := x + y, wp(y := x - y, wp(x := x - y, x = a \wedge y = b)))$

≡ Zuweisungsregel

$wp(x := x + y, wp(y := x - y, x - y = a \wedge y = b \wedge -128 \le x - y \le 127))$

≡ Zuweisungsregel, Arithmetik

$wp(x := x + y, y = a \wedge x - y = b \wedge -128 \le y \le 127 \wedge -128 \le x - y \le 127)$

≡ Zuweisungsregel, Arithmetik

$y = a \wedge x = b \wedge -128 \le y \le 127 \wedge -128 \le x \le 127 \wedge -128 \le x + y \le 127$

≡ Definition von x und y

$y = a \wedge x = b \wedge -128 \le x + y \le 127$

•

3.2.4 Fallunterscheidung

3.2.4.1 If-Statement

Die Ausführung des If-Statements **`if`** `b` **`then`** `P1` **`else`** `P2` **`end if`** (`IF`) besteht zunächst in der Auswertung des Booleschen Ausdrucks b. Falls diese Auswertung normal terminiert und *true* ergibt, wird $P1$ ausgeführt. Falls die Auswertung normal terminiert und *false* ergibt, wird $P2$ ausgeführt. Falls diese Auswertung nicht normal terminiert, terminiert auch das If-Statement nicht normal. Unter der Voraussetzung, daß die Auswertung des Booleschen Ausdrucks b normal terminiert, gilt somit: wenn IF im Zustand s gestartet wird und $b(s)$ gilt, ist die Endzustandsmenge $f_{P1}(s)$. Wenn IF im Zustand s gestartet wird und $b(s)$ nicht gilt, ist die Endzustandsmenge $f_{P2}(s)$. Wegen $b(s) \equiv (C^{-1}(b(s)) = Z)$ gilt also für die Programmfunktion

$$(\forall s \in Z: f_{IF}(s) = C^{-1}(b(s)) \cap f_{P1}(s) \cup \overline{C^{-1}(b(s))} \cap f_{P2}(s)).$$

Wenn $b(s)$ gilt, ergibt der Schnitt von $f_{P1}(s)$ mit Z $f_{P1}(s)$ und der Schnitt von $f_{P2}(s)$ mit $\overline{Z}$ $\varnothing$, so daß in diesem Fall $f_{IF}(s) = f_{P1}(s)$ ist (und umgekehrt).

Lemma 3.8 (If-Regel): Mit IF :≡ **`if`** `b` **`then`** `P1` **`else`** `P2` **`end if`**; und normaler Terminierung der Auswertung des Boolschen Ausdrucks b gilt

$$[wp(IF, N) \equiv b \wedge wp(P1, N) \vee \neg b \wedge wp(P2, N)].$$

Beweis: zu zeigen ist $LP(IF, M) = C^{-1}(b) \cap LP(P1, M) \cup C^{-1}(\neg b) \cap LP(P2, M)$

$LP(IF, M)$

= Definition LP

$\{s \in Z: f_{IF}(s) \subseteq M\}$

= Definition f_{IF}

$\{s \in Z: C^{-1}(b(s)) \cap f_{P1}(s) \cup \overline{C^{-1}(b(s))} \cap f_{P2}(s) \subseteq M\}$

= Z-split ($Z = C^{-1}(b(s)) \cup \overline{C^{-1}(b(s))}$)

$\{s \in C^{-1}(b(s)): C^{-1}(b(s)) \cap f_{P1}(s) \cup \overline{C^{-1}(b(s))} \cap f_{P2}(s) \subseteq M\} \cup$
$\{s \in \overline{C^{-1}(b(s))}: C^{-1}(b(s)) \cap f_{P1}(s) \cup \overline{C^{-1}(b(s))} \cap f_{P2}(s) \subseteq M\}$

= Mengenlehre, Logik

$\{s \in C^{-1}(b(s)): f_{P1}(s) \subseteq M\} \cup \{s \in \overline{C^{-1}(b(s))}: f_{P2}(s) \subseteq M\}$

= Mengenlehre, Logik

$C^{-1}(b) \cap \{s \in Z: f_{P1}(s) \subseteq M\} \cup \overline{C^{-1}(b(s))} \cap \{s \in Z: f_{P2}(s) \subseteq M\}$

= Definition LP

$C^{-1}(b) \cap LP(P1, M) \cup C^{-1}(\neg b) \cap LP(P2, M)$

•

Die normale Terminierung der Auswertung des Booleschen Ausdrucks b wird durch $bool(b)$ bezeichnet. Damit ergibt sich $wp(IF, N) \equiv bool(b) \wedge (b \wedge wp(P1, N) \vee \neg b \wedge wp(P2, N))$. Das If-Statement ohne **`else`** erhält man aus IF durch Ersetzung von P2 durch **`null`**. Damit erhält man

$wp(if\ b\ then\ P\ end\ if, N)$

≡ If-Regel

$bool(b) \wedge (b \wedge wp(P1, N) \vee \neg b \wedge wp(NULL, N))$

≡ Null-Regel

$bool(b) \wedge (b \wedge wp(P1, N) \vee \neg b \wedge N)$

Beispiele:

```
x: Int8;
  :
if 127/x < 0 then x := -x; end if;
-- x ≥ 0
```

$wp(IF, x \geq 0)$

≡ If-Regel, Zuweisungsregel

$bool(127 / x < 0) \wedge (127 / x < 0 \wedge -x \geq 0 \wedge -128 \leq -x \leq 127 \vee 127 / x \geq 0 \wedge x \geq 0)$

≡ Arithmetik

$x \neq 0 \wedge (x < 0 \wedge x \leq 0 \wedge 128 \geq x \geq -127 \vee x > 0 \wedge x \geq 0)$

≡ Arithmetik, Definition von x

$-127 \leq x < 0 \vee 0 < x \leq 127$

Das folgende Beispiel zeigt die Verifikation einer If-Anweisung mit Hilfe von FPP.

Eingabe:

```
--!pre: -127<=x and x <= 127 and x = Kx;
   if x<0 then x := -x; end if;
--!post: -128<=x and x <= 127 and x = Abs(Kx);
```

Ausgabe:

```
         FPP (Frege Program Prover) University of Jena, Germany
User: 141.35.14.243       At: 98-3-3, 10:55

The answer to your query is:
--!pre       : (-127 <= x AND x <= 127 AND x = kx)
--> wp       :     (0 >= 1 + x AND -128 <= -x AND -x <= 127 AND -x = Abs(kx))
-->              OR (0 <= x AND -128 <= x AND x <= 127 AND x = Abs(kx))
--> vc       :      (-127 <= x AND x <= 127 AND x = kx)
-->            ==>     (0 >= 1 + x AND -128 <= -x AND -x <= 127 AND -x = Abs(kx))
-->                    OR (0 <= x AND -128 <= x AND x <= 127 AND x = Abs(kx))
--> Result: proved
IF x < 0 THEN
   x := -x;
end if;
--!post      : (-128 <= x AND x <= 127 AND x = Abs(kx))
```

3.2.4.2 Case-Statement

Das Case-Statement CASE ist eine Verallgemeinerung des If-Statements. Es hat folgende Form (in Ada-Syntax):

```
case e is
  when C1 => P1;
        :
  when Cn => Pn;
end case;
```

Die C_i sind die Vergleichsausdrücke. Sie bestehen aus einer durch | getrennten Liste von statischen Ausdrücken bzw. statischen Bereichen:

```
Ci  ::= Di1 '|' ... '|' Din
Dij ::= Kij | UGij '..' OGij
```

e ist ein diskreter Ausdruck. K_{ij}, UG_{ij} und OG_{ij} sind statische Ausdrücke von gleichem Typ wie *e*. Zu jedem C_i gibt es den charakteristischen Vergleichsausdruck CVA_i. Er ist definiert durch $CVA_i :\equiv D_{i1} \vee \cdots \vee D_{in}$, wobei $D_{ij} :\equiv \begin{cases} e = K_{ij}, & falls\ D_{ij} \equiv K_{ij} \\ UG_{ij} \le e \le OG_{ij}, & falls\ D_{ij} \equiv UG_{ij} \ldots OG_{ij} \end{cases}$.

Die C_i bilden eine Partition des Typs von *e*, d.h. sie sind paarweise disjunkt und schöpfen den Typ von *e* aus, formal

$$(\forall s \in Z : CVA_1(s) \vee \cdots \vee CVA_n(s)) \wedge (\forall i, j : 1 \le i < j \le n : \neg(\exists s \in Z : CVA_i(s) \equiv CVA_j(s))).$$

Zur Ausführung von CASE wird zunächst *e* im aktuellen Zustand *s* ausgewertet. Falls die Auswertung von *e* normal terminiert und $CVA_i(s)$ gilt, wird P_i ausgeführt. Wegen der Partitionierung gilt immer genau ein $CVA_i(s)$. Falls die Auswertung von *e* nicht normal terminiert, terminiert auch CASE nicht normal. Unter der Voraussetzung, daß die Auswertung des Ausdrucks *e* normal terminiert, gilt somit (mit analoger Argumentation wie bei IF)

$$(\forall s \in Z : f_{CASE}(s) = C^{-1}(CVA_1) \cap f_{P1}(s) \cup \cdots \cup C^{-1}(CVA_n) \cap f_{Pn}(s))$$

Für ein Case-Statement mit **others** ist $C_n \equiv others$ und $CVA_n \equiv \neg(CVA_1 \vee \cdots \vee CVA_{n-1})$.

Lemma 3.9 (Case-Regel): Unter der Voraussetzung, daß die Auswertung des Ausdrucks *e* normal terminiert, gilt $wp(CASE, N) \equiv CVA_1 \wedge wp(P1, N) \vee \cdots \vee CVA_n \wedge wp(Pn, N)$.

Beweis: zu zeigen ist (mit $CVM_i := C^{-1}(CVA_i(s))$ (*charakteristische Vergleichsmenge*))

$LP(CASE, M) = C^{-1}(CVA_1) \cap LP(P1, M) \cup \cdots \cup C^{-1}(CVA_n) \cap LP(Pn, M)$.

$LP(CASE, M)$

= Definition *LP*

$\{s \in Z: f_{CASE}(s) \subseteq M\}$

= Definition f_{CASE}

$\{s \in Z: CVM_1 \cap f_{P1}(s) \cup \cdots \cup CVM_n \cap f_{Pn}(s) \subseteq M\}$

= *Z*-split

$\{s \in CVM_1: CVM_1 \cap f_{P1}(s) \cup \cdots \cup CVM_n \cap f_{Pn}(s) \subseteq M\} \cup$

$\vdots$

$\{s \in CVM_n: CVM_1 \cap f_{P1}(s) \cup \cdots \cup CVM_n \cap f_{Pn}(s) \subseteq M\}$

= Mengenlehre, Logik, CVM_i bilden Partition von *Z*

$\{s \in CVM_1: f_{P1}(s) \subseteq M\} \cup \cdots \cup \{s \in CVM_n: CVM_n \cap f_{Pn}(s) \subseteq M\}$

= Mengenlehre, Logik

$CVM_1 \cap \{s \in Z: f_{P1}(s) \subseteq M\} \cup \cdots \cup CVM_n \cap \{s \in Z: CVM_n \cap f_{Pn}(s) \subseteq M\}$

= Definition *LP*

$C^{-1}(CVA_1) \cap LP(P1, M) \cup \cdots \cup C^{-1}(CVA_n) \cap LP(Pn, M)$

•

Mit der Prüfung auf normale Terminierung der Auswertung von *e* ergibt sich

$wp(CASE, N) \equiv T_e(e) \wedge (CVA_1 \wedge wp(P1, N) \vee \cdots \vee CVA_n \wedge wp(Pn, N))$.

Beispiel:

```
x: Int8;
s: Nat8;
   :
case x is
   when -128 .. -1   => s := -1;
   when 0            => s := x;
   when 1 .. 127     => s := 1;
end case;
-- s*x ≥ 0
```

$wp(CASE, s * x \geq 0)$

≡ Case-Regel

$T(x) \wedge (-128 \leq x \leq -1 \wedge -x \geq 0 \wedge 0 \leq -1 \leq 127 \vee$

$x = 0 \wedge x^2 \geq 0 \wedge 0 \leq x \leq 127 \vee 1 \leq x \leq 127 \wedge x \geq 0 \wedge 0 \leq 1 \leq 127)$

≡ Arithmetik, Logik

$x = 0 \vee 1 \le x \le 127$

≡ Arithmetik, Logik

$0 \le x \le 127$

3.2.5 While-Schleifen

Die While-Schleife **`while`** `b` **`loop`** `P` **`end loop`** (LOOP) wird folgendermaßen ausgeführt: zunächst wird der Boolesche Ausdruck *b* ausgewertet. Falls die Auswertung normal terminiert und *false* ergibt, wird nichts gemacht. Falls die Auswertung normal terminiert und *true* ergibt, wird der Schleifenrumpf *P* ausgeführt und die ganze Schleife erneut ausgeführt. Damit ist [B95: 76]

`LOOP` ≡ **`if`** `b` **`then`** `P; LOOP;` **`end if`**; und mit $B := C^{-1}(b)$ gilt

$LP(LOOP, M)$

≡ If- und Sequenz-Regel

$\overline{B} \cap M \cup B \cap LP(P, LP(LOOP, M))$

Die unbekannte Menge $LP(LOOP, M)$ kommt in der Gleichung sowohl auf der linken Seite als auch implizit auf der rechten Seite als Teil eines Ausdrucks vor. $LP(LOOP, M)$ ist daher ein Fixpunkt der Funktion

$$g_M : 2^Z \mapsto 2^Z$$
$$g_M(S) = \overline{B} \cap M \cup B \cap LP(P, S)$$

unter der Voraussetzung, daß ein solcher existiert. In [B95: 77] wird die *wp* bzw. *LP* der While-Schleife als kleinster Fixpunkt *definiert*. Hier wird, wegen der prinzipiell anderen Vorgehensweise bei der Einführung der *wp*- bzw. *LP*-Funktion als in [B95] (siehe auch Einleitung von Abschnitt 3), dieses als Satz formuliert (Satz 3.16), der natürlich bewiesen werden muß. Die folgenden Lemmata werden zum Beweis dieses Satzes benötigt.

Lemma 3.10 (Fixpunkt): Die Funktion g_M besitzt einen Fixpunkt.
Beweis: Eine monotone Funktion auf einem vollständigen Verband besitzt einen Fixpunkt [B95: 30]. Da der Teilmengenverband $(2^Z, \subseteq, \cap, \cup, \bar{}, \varnothing, Z)$ vollständig ist, muß nur noch die Monotonie gezeigt werden, also $A \subseteq D \Rightarrow g_M(A) \subseteq g_M(D)$:

$g_M(A)$

= Definition g_M

$\overline{B} \cap M \cup B \cap LP(P, A)$

$\subseteq$ Voraussetzung, Monotonie von *LP*, Mengenlehre

$\overline{B} \cap M \cup B \cap LP(P, D)$

= Definition g_M

$g_M(D)$

•

Da die Menge der Fixpunkte selbst einen vollständigen Verband bildet [B95: 30], hat g_M auch einen kleinsten Fixpunkt μg. Falls g_M stetig ist, kann man μg konstruieren. Falls die Definitions- bzw. Wertemenge von g_M nur endliche Ketten enthält, ist die Stetigkeit gleichbedeutend mit der Monotonie [SL87: 77]. Da *Z* und damit 2^Z endlich ist, ist g_M stetig. Für den kleinsten Fixpunkt gilt dann $\mu g = \bigcup_{i \geq 0} g_M^i(\varnothing)$.

Lemma 3.11: $\{g_M^i(\varnothing), i \geq 0\}$ ist eine monotone endliche Kette.
Beweis: zu zeigen ist $(\forall i \geq 0: g_M^i(\varnothing) \subseteq g_M^{i+1}(\varnothing))$ durch Induktion über *i*:

$i = 0$: $g_M^0(\varnothing) = \varnothing \subseteq g_M^1(\varnothing)$

$i \to i+1$: unter der Induktionsvoraussetzung $g_M^i(\varnothing) \subseteq g_M^{i+1}(\varnothing)$ (die Aussage gilt für *i*) ist zu zeigen $g_M^{i+1}(\varnothing) \subseteq g_M^{i+2}(\varnothing)$ (die Aussage gilt für $i+1$).

$g_M^{i+1}(\varnothing)$

=

$g_M(g_M^i(\varnothing))$

= Definition g_M

$\overline{B} \cap M \cup B \cap LP(P, g_M^i(\varnothing))$

$\subseteq$ Monotonie

$\overline{B} \cap M \cup B \cap LP(P, g_M^{i+1}(\varnothing))$

= Definition g_M

$g_M^{i+2}(\varnothing)$

Die Endlichkeit der Kette wird durch die Endlichkeit von *Z* impliziert.

•

Daß μg ein Fixpunkt von g_M ist, wird explizit durch Einsetzen gezeigt.

Lemma 3.12: $g_M(\mu g) = \mu g$.
Beweis:

$g_M(\mu g)$

= Definition g_M

$\overline{B} \cap M \cup B \cap LP(P, \mu g)$

= Konstruktion μg

$\overline{B} \cap M \cup B \cap LP(P, \bigcup_{i \geq 0} g_M^i(\varnothing))$

= starke Disjunktivität bei endlichen Ketten (Lemma 2.37), Lemma 3.11,
Mengenlehre

$\bigcup_{i \geq 0} (\overline{B} \cap M \cup B \cap LP(P, g_M^i(\varnothing)))$

= Definition g_M

$\bigcup_{i\geq 0} g_M(g_M^i(\varnothing))$

=

$\bigcup_{i\geq 0} g_M^{i+1}(\varnothing)$

= Indexverschiebung

$\bigcup_{i\geq 1} g_M^i(\varnothing)$

= wegen $g_M^0(\varnothing) = \varnothing$

$\bigcup_{i\geq 0} g_M^i(\varnothing)$

= Konstruktion μg

μg

•

Zum Beweis, daß $LP(LOOP, M)$ der kleinste Fixpunkt ist und kein anderer, benötigt man noch einige Vorbereitungen. Zunächst wird Z in $m+2$ Teilmengen Z_k von Zuständen aufgeteilt, so daß *LOOP* nach Start in einem dieser Zustände nach höchstens *k* Iterationen terminiert. Wegen der Endlichkeit von *Z* kann es nur endlich viele Teilmengen geben. Die Zustände, die keine Terminierung garantieren, kommen in eine extra Menge Z_∞. Wegen Nichtdeterminismus ist es möglich, daß *LOOP* bei gleichem Startzustand nach verschieden vielen Iterationen terminiert oder einmal terminiert und einmal nicht.

Definition 3.13 (k-Zustände): $Z_k := \{s \in Z: f_{P^k}(s) \subseteq \overline{B} \cap M\}$, $0 \leq k \leq m$, $Z_\infty := Z \setminus \bigcup_{k=0}^{m} Z_k$.

•

Lemma 3.14: $Z_\infty \cup \bigcup_{k=0}^{m} Z_k = Z$.

Beweis:

$Z_\infty \cup \bigcup_{k=0}^{m} Z_k$

= Definition Z_∞

$(Z \setminus \bigcup_{k=0}^{m} Z_k) \cup \bigcup_{k=0}^{m} Z_k$

= Mengenlehre $(A \setminus B) \cup B = A \cup B$

$Z \cup \bigcup_{k=0}^{m} Z_k$

= wegen $Z_k \subseteq Z$

Z

•

Wenn *LOOP* in einem Zustand gestartet wird, der Terminierung nach höchstens $k+1$ Iterationen garantiert, dann ist nach einer Iteration die Terminierung nach höchstens *k* Iterationen garantiert und umgekehrt.

Lemma 3.15: $(\forall k: 0 \leq k \leq m-1: (\forall s: s \in Z_{k+1} \equiv s \in Z \wedge f_P(s) \subseteq Z_k))$.

Beweis:

$s \in Z_{k+1} \equiv s \in Z \wedge f_P(s) \subseteq Z_k$

≡ Definition Z_k, Mengenlehre

$s \in Z \wedge f_{P^{k+1}}(s) \subseteq \overline{B} \cap M \equiv s \in Z \wedge (\forall t \in Z : t \in f_P(s) \Rightarrow t \in Z_k)$

≡ Programmfunktion der Sequenz

$s \in Z \wedge \bigcup_{t \in f_P(s)} f_{P^k}(t) \subseteq \overline{B} \cap M \equiv s \in Z \wedge (\forall t \in Z : t \in f_P(s) \Rightarrow t \in Z_k)$

≡ Mengenlehre, Definition Z_k

$s \in Z \wedge (\forall t \in f_P(s) : f_{P^k}(t) \subseteq \overline{B} \cap M) \equiv s \in Z \wedge (\forall t \in f_P(s) \Rightarrow f_{P^k}(t) \subseteq \overline{B} \cap M)$

≡ Logik

true

•

Satz 3.16 (While-Regel): $LP(LOOP, M) = \mu g$.

Beweis: $\mu g \subseteq LP(LOOP, M)$ gilt, weil $LP(LOOP, M)$ ein Fixpunkt ist, die Menge der Fixpunkte ein vollständiger Verband ist und μg der kleinste Fixpunkt ist.

$LP(LOOP, M) \subseteq \mu g$: dieser Teil besagt, daß $LP(LOOP, M)$ tatsächlich der kleinste Fixpunkt ist und damit kein anderer sein kann. (Denn wäre $LP(LOOP, M)$ ein anderer Fixpunkt, müßte $LP(LOOP, M)$ eine echte Obermenge von μg sein. Aber eine Menge kann nicht gleichzeitig Teilmenge und echte Obermenge einer anderen Menge sein.)

$LP(LOOP, M) \subseteq \mu g$

≡ Definition von *LP*

$\{s \in Z : f_{LOOP}(s) \subseteq M\} \subseteq \mu g$

≡ *Z*-split, Lemma 3.14

$\bigcup_{k=0}^{m} \{s \in Z_k : f_{LOOP}(s) \subseteq M\} \cup \{s \in Z_\infty : f_{LOOP}(s) \subseteq M\} \subseteq \mu g$

≡ Mengenlehre, $\{s \in Z_\infty : f_{LOOP}(s) \subseteq M\} = \emptyset$

$(\forall k : 0 \le k \le m : \{s \in Z_k : f_{LOOP}(s) \subseteq M\} \subseteq \mu g)$

≡ Rekursive Definition von *LOOP* und μg

$(\forall k : 0 \le k \le m : \{s \in Z_k : f_{IF}(s) \subseteq M\} \subseteq \bigcup_{i \ge 0} g_M^i(\emptyset))$

≡ If-Regel

$(\forall k : 0 \le k \le m : \{s \in Z_k : \overline{C^{-1}(b(s))} \cap \{s\} \cup C^{-1}(b(s)) \cap f_{P;LOOP}(s) \subseteq M\} \subseteq \bigcup_{i \ge 0} g_M^i(\emptyset))$

≡ Sequenzregel

$(\forall k : 0 \le k \le m : \{s \in Z_k : \overline{C^{-1}(b(s))} \cap \{s\} \cup C^{-1}(b(s)) \cap \bigcup_{t \in f_P(s)} f_{LOOP}(t) \subseteq M\} \subseteq \bigcup_{i \ge 0} g_M^i(\emptyset))$

Das wird durch Induktion über *m* bewiesen.

$m = 0$:

$$\{s \in Z_0 : \overline{C^{-1}(b(s))} \cap \{s\} \cup C^{-1}(b(s)) \cap \bigcup_{t \in f_P(s)} f_{LOOP}(t) \subseteq M\} \subseteq \bigcup_{i \geq 0} g_M^i(\varnothing)$$

≡ Definition von Z_0 und g_M

$$\{s \in \overline{B} \cap M : \overline{C^{-1}(b(s))} \cap \{s\} \cup C^{-1}(b(s)) \cap \bigcup_{t \in f_P(s)} f_{LOOP}(t) \subseteq M\} \subseteq \overline{B} \cap M \cup \bigcup_{i \geq 2} g_M^i(\varnothing)$$

≡ Mengenlehre

true

$m > 0$:

$$(\forall k : 0 \leq k \leq m : \{s \in Z_k : \overline{C^{-1}(b(s))} \cap \{s\} \cup C^{-1}(b(s)) \cap \bigcup_{t \in f_P(s)} f_{LOOP}(t) \subseteq M\} \subseteq \mu g)$$

≡ Z_k-split, Mengenlehre, Logik

$$(\forall k : 0 \leq k \leq m : \{s \in Z_k \cap B : \overline{C^{-1}(b(s))} \cap \{s\} \cup C^{-1}(b(s)) \cap \bigcup_{t \in f_P(s)} f_{LOOP}(t) \subseteq M\} \subseteq \mu g) \wedge$$

$$(\forall k : 0 \leq k \leq m : \{s \in Z_k \cap \overline{B} : \overline{C^{-1}(b(s))} \cap \{s\} \cup C^{-1}(b(s)) \cap \bigcup_{t \in f_P(s)} f_{LOOP}(t) \subseteq M\} \subseteq \mu g)$$

≡ Mengenlehre

$$(\forall k : 0 \leq k \leq m : \{s \in Z_k \cap B : \bigcup_{t \in f_P(s)} f_{LOOP}(t) \subseteq M\} \subseteq \mu g) \wedge$$

$$(\forall k : 0 \leq k \leq m : \{s \in Z_0 \cap \overline{B} : \{s\} \subseteq M\} \subseteq \mu g)$$

≡ Induktionsvoraussetzung

$$(\forall k : 0 \leq k \leq m : \{s \in Z_k \cap B : \bigcup_{t \in f_P(s)} f_{LOOP}(t) \subseteq M\} \subseteq \mu g)$$

≡ Indexverschiebung, $\forall$-split

$$(\forall k : 0 \leq k \leq m-1 : \{s \in Z_{k+1} \cap B : \bigcup_{t \in f_P(s)} f_{LOOP}(t) \subseteq M\} \subseteq \mu g) \wedge \{s \in Z_0 \cap B : \bigcup_{t \in f_P(s)} f_{LOOP}(t) \subseteq M\} \subseteq \mu g$$

≡ mit $Z_0 \cap B = \varnothing$

$$(\forall k : 0 \leq k \leq m-1 : \{s \in Z_{k+1} \cap B : \bigcup_{t \in f_P(s)} f_{LOOP}(t) \subseteq M\} \subseteq \mu g)$$

≡ Mengenlehre

$$(\forall k : 0 \leq k \leq m-1 : \{s \in B : s \in Z_{k+1} \wedge \bigcup_{t \in f_P(s)} f_{LOOP}(t) \subseteq M\} \subseteq \mu g)$$

⇐ Lemma 3.15, Mengenlehre

$$(\forall k : 0 \leq k \leq m-1 : \{s \in B : f_p(s) \subseteq Z_k \wedge (\forall t \in f_P(s) : f_{LOOP}(t) \subseteq M)\} \subseteq \mu g)$$

≡ Mengenlehre

$$(\forall k : 0 \leq k \leq m-1 : (\forall x : x \in \{s \in B : f_p(s) \subseteq Z_k \wedge (\forall t \in f_P(s) : f_{LOOP}(t) \subseteq M)\} \Rightarrow x \in \mu g))$$

≡ Mengenlehre

$(\forall k : 0 \le k \le m-1 : (\forall x : x \in B \wedge f_p(x) \subseteq Z_k \wedge (\forall t \in f_P(x) : f_{LOOP}(t) \subseteq M) \Rightarrow x \in \mu g))$

⇐ Induktionsvoraussetzung für $f_{LOOP}(t) \subseteq M$ (wegen $t \in f_P(x)$, $f_p(x) \subseteq Z_k$ und $k < m$)

$(\forall k : 0 \le k \le m-1 : (\forall x : x \in B \wedge f_p(x) \subseteq Z_k \wedge (\forall t \in f_P(x) : t \in \mu g) \Rightarrow x \in \mu g))$

≡ Mengenlehre, Definition μg

$(\forall k : 0 \le k \le m-1 : (\forall x : x \in B \wedge f_p(x) \subseteq Z_k \wedge f_P(x) \subseteq \mu g \Rightarrow x \in \bigcup_{i \ge 0} g_M^i(\varnothing)))$

≡ Mengenlehre, $g_M^0(\varnothing) = \varnothing$

$(\forall k : 0 \le k \le m-1 : (\forall x : x \in B \wedge f_p(x) \subseteq Z_k \wedge f_P(x) \subseteq \mu g \Rightarrow x \in \bigcup_{i \ge 1} g_M(g_M^{i-1}(\varnothing))))$

≡ Mengenlehre, Definition g_M

$(\forall k : 0 \le k \le m-1 : (\forall x : x \in B \wedge f_p(x) \subseteq Z_k \wedge f_P(x) \subseteq \mu g \Rightarrow (\exists i \ge 1 : x \in \overline{B} \cap M \cup B \cap LP(P, g_M^{i-1}(\varnothing)))))$

≡ Mengenlehre, Logik, Definition g_M

$(\forall k : 0 \le k \le m-1 : (\forall x : x \in B \wedge f_p(x) \subseteq Z_k \wedge f_P(x) \subseteq \mu g \Rightarrow (\exists i \ge 1 : x \in LP(P, g_M^{i-1}(\varnothing)))))$

≡ Definition *LP*

$(\forall k : 0 \le k \le m-1 : (\forall x : x \in B \wedge f_p(x) \subseteq Z_k \wedge f_P(x) \subseteq \mu g \Rightarrow (\exists i \ge 1 : x \in Z \wedge f_P(x) \subseteq g_M^{i-1}(\varnothing))))$

≡ Mengenlehre, $B \subseteq Z$ wegen normaler Terminierung der Auswertung von *b*

$(\forall k : 0 \le k \le m-1 : (\forall x : x \in B \wedge f_p(x) \subseteq Z_k \wedge f_P(x) \subseteq \mu g \Rightarrow f_P(x) \subseteq \bigcup_{i \ge 1} g_M^{i-1}(\varnothing)))$

≡ Indexverschiebung, Definition g_M

$(\forall k : 0 \le k \le m-1 : (\forall x : x \in B \wedge f_p(x) \subseteq Z_k \wedge f_P(x) \subseteq \mu g \Rightarrow f_P(x) \subseteq \mu g))$

≡ Logik

true

•

Für die schwächste Vorbedingung der While-Schleife ergibt sich damit $wp(LOOP, N) \equiv \mu h$, wobei μh der kleinste Fixpunkt der Funktion $h_N(R) :\equiv \neg b \wedge N \vee b \wedge wp(P, R)$ ist. Man kann also die schwächste Vorbedingung dadurch ermitteln, daß man die Funktion h_N wiederholt anwendet, beginnend bei *false*, bis sich nichts mehr ändert bzw. bis man ein allgemeines Schema erkennt. Auf Grund dieser Formulierung ergibt sich schon, daß dieses Verfahren nicht immer funktioniert. Daher wird zum Korrektheitsbeweis von Schleifen nicht die schwächste Vorbedingung, sondern eine VC verwendet, die aus einer Schleifeninvarianten und einer Terminierungsfunktion gebildet wird (siehe Abschnitt 3.4.1). In Abschnitt 4.2 wird ein Verfahren vorgestellt, das versucht, die schwächste Vorbedingung direkt zu ermitteln.

3.2.6 Nichtdeterministische Auswahl

Das Programm `NonDet P1 ◊ ... ◊ Pn` führt genau eins der Programme `P1 ... Pn` aus. Die Endzustandsmenge eines Zustands, in dem NonDet gestartet wird, ist daher die Endzustandsmenge von *P*1 oder Endzustandsmenge von *P*2 oder ... oder Endzustandsmenge von *Pn*. Für die Programmfunktion ergibt sich damit

$$(\forall s \in Z : f_{NonDet}(s) = \bigcup_{i=1}^{n} f_{Pi}(s)) .$$

Lemma 3.17 (NonDet-Regel): $wp(P_1 \Diamond \cdots \Diamond P_n, N) \equiv (\forall i{:}1 \le i \le n{:} wp(P_i, N))$.

Beweis: zu zeigen ist $LP(P_1 \Diamond \cdots \Diamond P_n, M) = \bigcap_{i=1}^{n} LP(P_i, M)$

$LP(P_1 \Diamond \cdots \Diamond P_n, M)$

= Definition *LP*

$\{s \in Z{:} f_{NonDet}(s) \subseteq M\}$

= Definition f_{NonDet}

$\{s \in Z{:} \bigcup_{i=1}^{n} f_{Pi}(s) \subseteq M\}$

= Mengenlehre

$\{s \in Z{:} (\forall i{:}1 \le i \le n{:} f_{Pi}(s) \subseteq M)\}$

zu zeigen bleibt

$$\{s \in Z{:} (\forall i{:}1 \le i \le n{:} f_{Pi}(s) \subseteq M)\} = \bigcap_{i=1}^{n} \{s \in Z{:} f_{Pi}(s) \subseteq M)$$

≡ Mengenlehre

$$(\forall x{:} x \in \{s \in Z{:} (\forall i{:}1 \le i \le n{:} f_{Pi}(s) \subseteq M)\} \equiv x \in \bigcap_{i=1}^{n} \{s \in Z{:} f_{Pi}(s) \subseteq M\})$$

≡ Mengenlehre

$$(\forall x : x \in Z \wedge (\forall i : 1 \le i \le n : f_{Pi}(x) \subseteq M) \equiv (\forall i : 1 \le i \le n : x \in Z \wedge f_{Pi}(x) \subseteq M))$$

≡ Logik

true

•

Bemerkung: Auch hier sieht man wieder, wie rein formal hergeleitet wurde, was vielleicht intuitiv nicht so einleuchtend ist: wenn nach der Ausführung von *P*1 ∨ ... ∨ *Pn N* gelten soll, muß vorher $wp(P_1, N) \wedge ... \wedge wp(P_n, N)$ gelten.

3.3 Substitutionslemma für *wp*

An dieser Stelle wird etwas vom eigentlichen Thema dieses Kapitels abgewichen. Es wird dargestellt, unter welchen Voraussetzungen Lemma 2.9 auch für die Funktion *wp* formuliert werden kann. Das sind die Aussagen der beiden folgenden Lemmata. Die Beweise werden jeweils durch Induktion über den Aufbau der Statements geführt und benutzen die *wp*s der entsprechenden Statements. Aus diesem Grund werden die beiden Lemmata an dieser Stelle gebracht.
Lemma 2.9 gilt für *wp* dann, wenn die zu substituierende Variable nicht verändert wird und die durch das Programm *S* veränderten Variablen nicht im substituierten Ausdruck vorkommen.

Lemma 3.18 (Substitutionslemma für *wp*): Sei *S* ein Programm, *N* ein Prädikat, *x* und *v* Variablen sowie *e* ein Ausdruck. Dann gilt

$$[trans(S,v) \wedge (\forall x : \neg trans(S,x) : x \notin free(e)) \Rightarrow (wp(S,N)_e^v \equiv wp(S_e^v, N_e^v))].$$

Beweis:
Die Bedingung $trans(S,v)$ ist notwendig, da eine Variable, die geändert wird, nicht durch einen Ausdruck ersetzt werden kann, *bevor* sie geändert wird (S_e^v ist dann im Allgemeinen syntaktisch falsch und damit wird der ganze Ausdruck undefiniert. Falls der substituierte Ausdruck eine Variable ist, kann Lemma 3.19 angewendet werden.). Aber diese Bedingung alleine reicht noch nicht. Der Beweis für die Notwendigkeit der anderen Bedingung erfolgt über den induktiven Aufbau von *S*. Den Induktionsanfang bilden das Null-Statement und die Wertzuweisung. Diese bildet auch den interessantesten Fall, weil dadurch Variablen verändert werden. Dieser Teil des Beweises ist auch der aufwendigste. Das ist ein Beispiel für einen Induktionsbeweis, bei dem der Induktionsanfang schwieriger als der Induktionsschritt ist.

$S \equiv Null$:

$trans(Null,v) \wedge (\forall x : \neg trans(Null,x) : x \notin free(e)) \Rightarrow (wp(Null,N)_e^v \equiv wp(Null_e^v, N_e^v))$

≡ Transparenz, Null-Regel

$[true \wedge (\forall x : false : x \notin free(e)) \Rightarrow (N_e^v \equiv N_e^v)]$

≡ Logik

true

$S \equiv z := f$, *z* ist eine Variable und *f* ein Ausdruck: Zu zeigen ist

$[trans(z := f, v) \wedge (\forall x : \neg trans(z := f, x) : x \notin free(e)) \Rightarrow (wp(z := f, N)_e^v \equiv wp((z := f)_e^v, N_e^v))]$

.

Fall $z = v$ ($z = v$ bedeutet hier Gleichheit der Namen der Variablen *z* und *v*, nicht Gleichheit der Werte; formal: $z = v :\equiv z \in free(v) \wedge v \in free(z)$):

$[trans(z := f, v) \wedge (\forall x : \neg trans(z := f, x) : x \notin free(e)) \Rightarrow (wp(z := f, N)_e^v \equiv wp((z := f)_e^v, N_e^v))]$

≡ Voraussetzung $z = v$

$[trans(v := f, v) \wedge (\forall x : \neg trans(v := f, x) : x \notin free(e)) \Rightarrow (wp(v := f, N)_e^v \equiv wp((v := f)_e^v, N_e^v))]$

≡ Transparenz

$[false \wedge (\forall x : \neg trans(v := f, x) : x \notin free(e)) \Rightarrow (wp(v := f, N)_e^v \equiv wp((v := f)_e^v, N_e^v))]$

≡ Logik

true

Fall $z \neq v$:

$[trans(z := f, v) \wedge (\forall x : \neg trans(z := f, x) : x \notin free(e)) \Rightarrow (wp(z := f, N)_e^v \equiv wp((z := f)_e^v, N_e^v))]$

≡ Voraussetzung $z \neq v$

$[trans(z := f, v) \wedge (\forall x : \neg trans(z := f, x) : x \notin free(e)) \Rightarrow (wp(z := f, N)_e^v \equiv wp(z := f_e^v, N_e^v))]$

≡ Logik

$[trans(z := f, v) \wedge (z \in Var(e) \vee z \notin free(e)) \wedge (\forall x : \neg trans(z := f, x) : x \notin free(e))$
$\Rightarrow (wp(z := f, N)_e^v \equiv wp(z := f_e^v, N_e^v))]$

≡ Distributivität, Logik

$[(trans(z := f, v) \wedge z \in free(e) \wedge (\forall x : \neg trans(z := f, x) : x \notin free(e)) \Rightarrow (wp(z := f, N)_e^v \equiv wp(z := f_e^v, N_e^v)))$
$(trans(z := f, v) \wedge z \notin free(e) \wedge (\forall x : \neg trans(z := f, x) : x \notin free(e)) \Rightarrow (wp(z := f, N)_e^v \equiv wp(z := f_e^v, N_e^v)))]$
≡ ∀-split

$[(trans(z := f, v) \wedge z \in free(e) \wedge (\forall x : x = z \wedge \neg trans(z := f, x) : x \notin free(e)) \wedge$
$(\forall x : x \neq z \wedge \neg trans(z := f, x) : x \notin free(e)) \Rightarrow (wp(z := f, N)_e^v \equiv wp(z := f_e^v, N_e^v))) \wedge$
$(trans(z := f, v) \wedge z \notin free(e) \wedge (\forall x : \neg trans(z := f, x) : x \notin free(e)) \Rightarrow (wp(z := f, N)_e^v \equiv wp(z := f_e^v, N_e^v)))]$
≡ one-point rule

$[(trans(z := f, v) \wedge z \in free(e) \wedge (\neg trans(z := f, z) \Rightarrow z \notin free(e)) \wedge$
$(\forall x : x \neq z \wedge \neg trans(z := f, x) : x \notin free(e)) \Rightarrow (wp(z := f, N)_e^v \equiv wp(z := f_e^v, N_e^v))) \wedge$
$(trans(z := f, v) \wedge z \notin free(e) \wedge (\forall x : \neg trans(z := f, x) : x \notin free(e)) \Rightarrow (wp(z := f, N)_e^v \equiv wp(z := f_e^v, N_e^v)))]$
≡ Transparenz

$[(trans(z := f, v) \wedge z \in free(e) \wedge (\neg false \Rightarrow z \notin free(e)) \wedge$
$(\forall x : x \neq z \wedge \neg trans(z := f, x) : x \notin free(e)) \Rightarrow (wp(z := f, N)_e^v \equiv wp(z := f_e^v, N_e^v))) \wedge$
$(trans(z := f, v) \wedge z \notin free(e) \wedge (\forall x : \neg trans(z := f, x) : x \notin free(e)) \Rightarrow (wp(z := f, N)_e^v \equiv wp(z := f_e^v, N_e^v)))]$
≡ Logik

$[(trans(z := f, v) \wedge false \wedge (\forall x : x \neq z \wedge \neg trans(z := f, x) : x \notin free(e)) \Rightarrow (wp(z := f, N)_e^v \equiv wp(z := f_e^v, N_e^v)))$
$(trans(z := f, v) \wedge z \notin free(e) \wedge (\forall x : \neg trans(z := f, x) : x \notin free(e)) \Rightarrow (wp(z := f, N)_e^v \equiv wp(z := f_e^v, N_e^v)))]$
≡ Logik

$[true \wedge (trans(z := f, v) \wedge z \notin free(e) \wedge (\forall x : \neg trans(z := f, x) : x \notin free(e))$
$\Rightarrow (wp(z := f, N)_e^v \equiv wp(z := f_e^v, N_e^v)))]$

≡ Logik, Zuweisungs-Regel

$[trans(z := f, v) \wedge z \notin free(e) \wedge (\forall x : \neg trans(z := f, x) : x \notin free(e)) \Rightarrow ((N_f^z)_e^v \equiv (N_e^v)_{f_e^v}^z)]$

≡ Substitutionslemma

$[trans(z := f, v) \wedge z \notin free(e) \wedge (\forall x : \neg trans(z := f, x) : x \notin free(e)) \Rightarrow (N_{f_e^v, e}^{z, v} \equiv N_{e_{f_e^v}^z, f_e^v}^{v, z})]$

≡ wegen $z \notin Var(e)$

$[trans(z := f, v) \wedge z \notin free(e) \wedge (\forall x : \neg trans(z := f, x) : x \notin free(e)) \Rightarrow (N_{f_e^v, e}^{z, v} \equiv N_{e, f_e^v}^{v, z})]$

≡ Logik

true

Induktionsschritt:
Unter der Annahme, daß das Lemma für *S*1,..., *Sn* gilt, ist zu zeigen, daß es dann auch für aus *S*1,..., *Sn* zusammengesetzte Anweisungen gilt. Das muß für alle zusammengesetzten Anwei-

sungen durchgeführt werden, was äußerst langwierig, aber nicht schwierig ist. Daher wird dieser Teil weggelassen.

•

Lemma 3.19 (Substitutionslemma für *wp*): Für eine Variable w mit $w \notin Var(S) \cup free(N)$ gilt $[wp(S,N)^g_w \equiv wp(S^g_w, N^g_w)]$.

Beweis: analog zum vorigen Beweis; es wird nur der interessante Fall der Zuweisung x := e betrachtet und danach unterschieden, ob die Variable x dieselbe wie g ist oder nicht.

Fall $x = g$: zu zeigen ist $wp(g := e, N)^g_w \equiv wp((g := e)^g_w, N^g_w)$

$wp(g := e, N)^g_w$

≡ Zuweisungsregel

$(N^g_e)^g_w$

≡ Substitutionslemma 2.12

$N^g_{e^g_w}$

≡ Substitutionslemma 2.12, $w \notin free(N)$

$(N^g_w)^w_{e^g_w}$

≡ Zuweisungsregel

$wp(w := e^g_w, N^g_w)$

≡

$wp((g := e)^g_w, N^g_w)$

Fall $x \neq g$: zu zeigen ist $wp(x := e, N)^g_w \equiv wp((x := e)^g_w, N^g_w)$

$wp(x := e, N)^g_w$

≡ Zuweisungsregel

$(N^x_e)^g_w$

≡ Substitutionslemma

$N^{x,g}_{e^g_w, w}$

≡ Substitutionslemma, $x \neq w$ (falls $x = w$, wäre $w \in Var(S)$ im Gegensatz zur Voraussetzung)

$(N^g_w)^x_{e^g_w}$

≡ Zuweisungsregel

$wp(x := e^g_w, N^g_w)$

≡ wegen $x \neq g$

$wp((x := e)^g_w, N^g_w)$

•

3.4 Approximierte Verifikationsbedingungen

3.4.1 While-Schleifen

Es werden nur terminierende While-Schleifen betrachtet. Für die Verifikation von While-Schleifen verwendet man eine Invariante *J* und eine Terminierungsfunktion *t*. *J* ist ein Prädikat, das an einer bestimmten Stelle im Schleifenrumpf bei jeder Iteration gilt (o.B.d.A. am Anfang des Rumpfes), sowie vor Betreten und nach Verlassen der Schleife. *t* ist eine nach unten beschränkte Funktion der Programmvariablen, die mit jeder Iteration kleiner wird, d.h. *t* ist eine Funktion von der Menge *W* der Werte der Variablen (Definition 2.15) in die natürlichen Zahlen, also $t \in \mathbf{N}^W$. Dadurch wird die Terminierung garantiert. (In der Mathematik gibt es zwar unendliche absteigende Ketten auf $\mathbb{R}$ und $\mathbb{Z}$, doch nicht auf dem endlichen Bereich der Zahlen im Rechner). Die annotierte While-Schleife sieht dann so aus:

```
-- Pre: V
-- Inv: J
-- Term: t
while b loop
   -- J ∧ b ∧ t = T
   P
   -- J ∧ t < T
end loop;
-- ¬b ∧ J
-- Post: N
```

Daraus ergibt sich, daß die VC die Konjunktion der folgenden 5 Implikationen ist [G81: 145]:

$[V \Rightarrow J]$

$[J \wedge b \Rightarrow wp(P, J)]$

$[J \wedge \neg b \Rightarrow N]$

$[J \wedge b \Rightarrow t \geq 0]$

$[J \wedge b \wedge t = T \Rightarrow wp(P, t < T)]$, wobei *T* eine neue Variable ist

Diese Bedingung ist nach dem Substitutionslemma äquivalent zu $[J \wedge b \Rightarrow wp(P, t < T)_t^T]$.

Bemerkung: $[J \wedge b \Rightarrow wp(P, t < T)_t^T]$ kann nicht durch Substitution von *t* für *T* vereinfacht werden, da die Voraussetzung des Substitutionslemmas für *wp* nicht erfüllt ist, denn *t* wird durch *P* verändert (sonst gäbe es keine Terminierung). Dann muß es mindestens 1 Variable in *t* geben, die durch *P* verändert wird, also $(\exists v : v \in free(t) : \neg trans(P, v))$, was äquivalent zu $\neg(\forall v : \neg trans(P, v) : v \notin free(t))$ ist.
Wenn man substituieren könnte, bekäme man als 5. Bedingung

$(J \wedge b \Rightarrow wp(P_t^T, t < t)) \equiv (J \wedge b \Rightarrow wp(P_t^T, false)) \equiv (J \wedge b \Rightarrow false) \equiv \neg(J \wedge b)$.

•

Daß diese 5 Bedingungen wirklich die Korrektheit der Schleife implizieren, muß natürlich beweisen werden. Für den Beweis braucht man das *Invarianztheorem* (Main Repetition Theorem [DS90: 180-182]). Es besagt folgendes: falls *t* nach unten beschränkt ist und mit jedem Schleifendurchlauf kleiner wird und *J* innerhalb der Schleife invariant ist, so gilt: nach Start in einem Zustand, in dem *J* erfüllt ist, terminiert die Schleife in einem Zustand, in dem *J* erfüllt ist und *b* nicht. Das Invarianztheorem ist der wesentliche Bestandteil des Beweises. Es wird hier einfach verwendet, ohne daß der Beweis noch einmal geführt wird.

Satz 3.20 (Invarianztheorem):

$[J \wedge b \Rightarrow t \geq 0] \wedge (\forall T{:}[J \wedge b \wedge t = T \Rightarrow wp(P, J \wedge t < T)]) \Rightarrow [J \Rightarrow wp(LOOP, J \wedge \neg b)]$

Satz 3.21 (While-Korrektheit):

$$(\exists J \in Rp(PS), t \in \mathbf{N}^W : [V \Rightarrow J] \wedge [J \wedge \neg b \Rightarrow N] \wedge$$
$$[J \wedge b \Rightarrow wp(P, J)] \wedge [J \wedge b \Rightarrow t \geq 0] \wedge [J \wedge b \wedge t = T \Rightarrow wp(P, t < T)]) \Rightarrow$$
$$[V \Rightarrow wp(LOOP, N)]$$

Bemerkung: J und t müssen gefunden werden. Wenn es ein Prädikat J und eine Funktion t mit den geforderten Eigenschaften gibt, ist die Schleife korrekt. Daher müssen J und t existenzquantifiziert werden.

Beweis:

$(\exists J \in Rp(PS), t \in \mathbf{N}^W : [V \Rightarrow J] \wedge [J \wedge \neg b \Rightarrow N] \wedge$
$[J \wedge b \Rightarrow wp(P, J)] \wedge [J \wedge b \Rightarrow t \geq 0] \wedge [J \wedge b \wedge t = T \Rightarrow wp(P, t < T)]) \Rightarrow$
$[V \Rightarrow wp(LOOP, N)]$

$\equiv$ Logik $((\exists x : P(x)) \Rightarrow R) \equiv (\forall x : P(x) \Rightarrow R)$

$(\forall J \in Rp(PS), t \in \mathbf{N}^W : [V \Rightarrow J] \wedge [J \wedge \neg b \Rightarrow N] \wedge$
$[J \wedge b \Rightarrow wp(P, J)] \wedge [J \wedge b \Rightarrow t \geq 0] \wedge [J \wedge b \wedge t = T \Rightarrow wp(P, t < T)] \Rightarrow$
$[V \Rightarrow wp(LOOP, N)])$

$\Leftarrow$ Logik $(\forall x : P(x)) \wedge (\forall x : R(x)) \equiv (\forall x : P(x) \wedge R(x))$,

$$((\forall x : P(x)) \Rightarrow (\forall x : R(x))) \Leftarrow (\forall x : P(x) \Rightarrow R(x))$$

$(\forall J \in Rp(PS), t \in \mathbf{N}^W : [(V \Rightarrow J) \wedge (J \wedge \neg b \Rightarrow N) \wedge$
$(J \wedge b \Rightarrow wp(P, J)) \wedge (J \wedge b \Rightarrow t \geq 0) \wedge (J \wedge b \wedge t = T \Rightarrow wp(P, t < T)) \Rightarrow$
$(V \Rightarrow wp(LOOP, N))])$

$\equiv$ Portation

$(\forall J \in Rp(PS), t \in \mathbf{N}^W : [(V \Rightarrow J) \wedge (J \wedge \neg b \Rightarrow N) \wedge$
$(J \wedge b \Rightarrow wp(P, J)) \wedge (J \wedge b \Rightarrow t \geq 0) \wedge (J \wedge b \wedge t = T \Rightarrow wp(P, t < T)) \wedge V \Rightarrow$
$wp(LOOP, N)])$

$\equiv$ Modus ponens

$(\forall J \in Rp(PS), t \in \mathbf{N}^W : [V \wedge J \wedge (J \wedge \neg b \Rightarrow N) \wedge$
$(J \wedge b \Rightarrow wp(P, J)) \wedge (J \wedge b \Rightarrow t \geq 0) \wedge (J \wedge b \wedge t = T \Rightarrow wp(P, t < T)) \Rightarrow$
$wp(LOOP, N)])$

$\Leftarrow$ Logik

$(\forall J \in Rp(PS), t \in \mathbf{N}^W : [V \wedge J \wedge (J \wedge \neg b \Rightarrow N) \wedge$
$(J \wedge b \wedge t = T \Rightarrow wp(P, J)) \wedge (J \wedge b \Rightarrow t \geq 0) \wedge (J \wedge b \wedge t = T \Rightarrow wp(P, t < T)) \Rightarrow$
$wp(LOOP, N)])$

$\equiv$ Konjunktivität

$(\forall J \in Rp(PS), t \in \mathbf{N}^W : [V \wedge J \wedge (J \wedge \neg b \Rightarrow N) \wedge$
$(J \wedge b \wedge t = T \Rightarrow wp(P, J \wedge t < T)) \wedge (J \wedge b \Rightarrow t \geq 0) \Rightarrow$
$wp(LOOP, N)])$

$\Leftarrow$ Invarianztheorem

$$[(J \wedge b \wedge t = T \Rightarrow wp(P, J \wedge t < T)) \wedge (J \wedge b \Rightarrow t \geq 0)] \Rightarrow [J \Rightarrow wp(LOOP, J \wedge \neg b)]$$

$(\forall J \in Rp(PS), t \in \mathbf{N}^W : [V \wedge J \wedge (J \wedge \neg b \Rightarrow N) \wedge (J \Rightarrow wp(LOOP, J \wedge \neg b) \Rightarrow$
$wp(LOOP, N)])$

$\equiv$ Modus ponens

$(\forall J \in Rp(PS), t \in \mathbf{N}^W : [V \wedge J \wedge (J \wedge \neg b \Rightarrow N) \wedge wp(LOOP, J \wedge \neg b) \Rightarrow wp(LOOP, N)])$

$\Leftarrow$ Monotonie von wp

$(\forall J \in Rp(PS), t \in \mathbf{N}^W : [V \wedge J \wedge (wp(LOOP, J \wedge \neg b) \Rightarrow wp(LOOP, N)) \wedge$
$wp(LOOP, J \wedge \neg b) \Rightarrow wp(LOOP, N)])$

$\equiv$ Modus ponens

$(\forall J \in Rp(PS), t \in \mathbf{N}^W : [V \wedge J \wedge wp(LOOP, N) \wedge wp(LOOP, J \wedge \neg b) \Rightarrow wp(LOOP, N)])$

$\equiv$ Logik

true

•

Daß zum Korrektheitsbeweis tatsächlich alle 5 Kriterien notwendig sind, sieht man an den folgenden Beispielen nicht korrekter Schleifen. In jedem Beispiel kann man 4 Kriterien zeigen, das 5. nicht.

1. *t* ist nicht nach unten beschränkt

```
--! Pre: i = 1;
--! Inv: true;
--! Term: - i;
while i > 0 loop i := i + 1; end loop;
--! Post: i <= 0;
```

Die Schleife terminiert nicht und ist damit nicht korrekt. Alle Kriterien außer der Beschränktheit von *t* können gezeigt werden.

- *t* ist nach unten beschränkt: $[i > 0 \wedge true \Rightarrow -i > 0] \equiv false$
- *t* nimmt ab: $[i > 0 \Rightarrow wp(t1 := -i; i := i + 1, -i < t1)] \equiv [i > 0 \Rightarrow -(i + 1) < -i] \equiv true$.
- *I* gilt vor Eintritt der Schleife: $[i = 1 \Rightarrow true] \equiv true$.
- *I* gilt nach Verlassen der Schleife: $[true \wedge \neg i > 0 \Rightarrow i \leq 0] \equiv true$.
- *I* ist invariant: $[true \wedge i > 0 \Rightarrow wp(i := i + 1, true)] \equiv [i > 0 \Rightarrow true] \equiv true$.

2. *t* nimmt nicht ab.

```
--! Pre: i = 1;
--! Inv: true;
--! Term: i;
while i > 0 loop i := i + 1; end loop;
--! Post: i <= 0;
```

Die Schleife terminiert nicht und ist damit nicht korrekt. Alle Kriterien außer der Abnahme von *t* können gezeigt werden.

- *t* ist nach unten beschränkt: $[i > 0 \wedge true \Rightarrow i > 0] \equiv true$.
- *t* nimmt ab: $[true \wedge i > 0 \Rightarrow wp(t1 := i; i := i + 1, i < t1)] \equiv [i > 0 \Rightarrow i + 1 < i] \equiv false$
- *I* gilt vor Eintritt der Schleife: $[i = 1 \Rightarrow true] \equiv true$.
- *I* gilt nach verlassen der Schleife: $[true \wedge \neg i > 0 \Rightarrow i \leq 0] \equiv true$.
- *I* ist invariant: $[true \wedge i > 0 \Rightarrow wp(i := i + 1, true)] \equiv [i > 0 \Rightarrow true] \equiv true$.

3. *I* gilt nicht vor Eintritt der Schleife.

```
--! Pre: i = 1;
--! Inv: i < 0;
--! Term: i;
while i > 0 loop i := i - 1; end loop;
--! Post: i < 0;
```

Die Schleife ist nicht korrekt, da nach der Terminierung nicht die Nachbedingung gilt. Alle Kriterien außer der Gültigkeit von *I* vor Eintritt der Schleife können gezeigt werden.

- *t* ist nach unten beschränkt: $[i > 0 \wedge i < 0 \Rightarrow i > 0] \equiv [\, false \Rightarrow i > 0] \equiv true$.
- *t* nimmt ab:
  $[i > 0 \wedge i < 0 \Rightarrow wp(t1 := i; i := i - 1, i < t1)] \equiv$
  $[\, false \Rightarrow wp(t1 := i; i := i - 1, i < t1)] \equiv true$
- *I* gilt vor Eintritt der Schleife: $[i = 1 \Rightarrow i < 0] \equiv false$
- *I* gilt nach verlassen der Schleife: $[i < 0 \wedge \neg i > 0 \Rightarrow i < 0] \equiv true$.
- *I* ist invariant:
  $[i < 0 \wedge i > 0 \Rightarrow wp(i := i - 1, i < 0)] \equiv [\, false \Rightarrow wp(i := i - 1, i < 0)] \equiv true$.

4. Nach Verlassen der Schleife gilt *I* nicht.

```
--! Pre: i = 1;
--! Inv: i <= 1;
--! Term: i;
while i > 0 loop i := i - 1; end loop;
--! Post: i < 0;
```

Die Schleife ist nicht korrekt, da nach der Terminierung nicht die Nachbedingung gilt. Alle Kriterien außer der Gültigkeit von *I* nach Verlassen der Schleife können gezeigt werden.

- *t* ist nach unten beschränkt: $[i > 0 \wedge i \leq 1 \Rightarrow i > 0] \equiv [i = 1 \Rightarrow i > 0] \equiv true$.
- *t* nimmt ab: $[i > 0 \wedge i \leq 1 \Rightarrow wp(t1 := i; i := i - 1, i < t1)] \equiv [i = 1 \Rightarrow i - 1 < i] \equiv true$
- *I* gilt vor Eintritt der Schleife: $[i = 1 \Rightarrow i \leq 1] \equiv true$
- *I* gilt nach verlassen der Schleife: $[i \leq 1 \wedge \neg i > 0 \Rightarrow i < 0] \equiv false$
- *I* ist invariant: $[i \leq 1 \wedge i > 0 \Rightarrow wp(i := i - 1, i \leq 1)] \equiv [i = 1 \Rightarrow i - 1 \leq 1] \equiv true$

5. *I* ist nicht invariant.

```
--! Pre: i = 1;
--! Inv: i <> 0 ;
--! Term: i;
while i > 0 loop i := i - 1; end loop;
--! Post: i < 0;
```

Die Schleife ist nicht korrekt, da nach der Terminierung nicht die Nachbedingung gilt. Alle Kriterien außer der Invarianz von I können gezeigt werden.

- t ist nach unten beschränkt: $[i > 0 \wedge i \neq 0 \Rightarrow i > 0] \equiv true$.
- t nimmt ab: $[i > 0 \wedge i \neq 0 \Rightarrow wp(t1 := i; i := i - 1, i < t1)] \equiv [i > 0 \Rightarrow i - 1 < i] \equiv true$.
- I gilt vor Eintritt der Schleife: $[i = 1 \Rightarrow i \neq 0] \equiv true$.
- I gilt nach Verlassen der Schleife: $[i \neq 0 \wedge \neg i > 0 \Rightarrow i < 0] \equiv true$.
- I ist invariant: $[i \neq 0 \wedge i > 0 \Rightarrow wp(i := i - 1, i \neq 0)] \equiv [i > 0 \Rightarrow i \neq 1] \equiv false$

•

Satz 3.21 besagt, daß eine Schleife korrekt ist, wenn es ein Prädikat J und eine Funktion t mit den geforderten Eigenschaften gibt. Das ist eine Implikation, bei der sich die Frage stellt, ob auch die Umkehrung gilt, und wenn nein, warum nicht. Hier gilt die Umkehrung fast, d.h. bis auf eine in der Praxis unbedeutende Ausnahme, die man aber beachten muß. Die Ausnahme, die ausgeschlossen werden muß, besteht darin, daß die Vorbedingung konstant *false* ist. Nach Definition der Korrektheit ist jedes Programm mit einer solchen Vorbedingung korrekt, d.h. auch nichtterminierende Schleifen.

Beispiel:

```
-- false
while true loop
   null;
end loop;
-- N
```

So eine Schleife ist zwar korrekt, aber nicht terminierend. Damit lautet die Umkehrung von Satz 3.21 verbal: zu jeder korrekten Schleife gibt es eine Invariante und eine Terminierungsfunktion unter der Voraussetzung, daß die Vorbedingung V nicht *false* ist, d.h. $C^{-1}(V) \neq \emptyset$. Diese Voraussetzung gilt für den Rest dieses Abschnitts. Eine Invariante ist $wp(LOOP, N)$ und eine Terminierungsfunktion ist die Anzahl der noch zu durchlaufenden Iterationen, die bei einer korrekten Schleife endlich und nicht negativ ist, und bei jeder Iteration verringert wird. Das ist die Aussage des folgenden Satzes, der bis auf die oben erwähnte Ausnahme genau die Umkehrung von Satz 3.21 ist. Zum Beweis benötigt man, daß $wp(LOOP, N)$ innerhalb der Schleife invariant ist [F93], und folgende Lemmata:

Lemma 3.22: $[wp(LOOP, N) \wedge \neg b \Rightarrow N]$.

Beweis: zu zeigen ist $LP(LOOP, M) \cap \overline{B} \subseteq M$.

$LP(LOOP, M) \cap \overline{B} \subseteq M$

≡ While-Regel, kleinster Fixpunkt

$\bigcup_{i \geq 0} g_M^i(\emptyset) \cap \overline{B} \subseteq M$

≡ Mengenlehre

$(\forall x \in Z : x \in \bigcup_{i \geq 0} g_M^i(\emptyset) \cap \overline{B} \Rightarrow x \in M)$

≡ Mengenlehre

$(\forall x \in Z : (\exists i \geq 0 : x \in g_M^i(\emptyset)) \wedge x \in \overline{B} \Rightarrow x \in M)$

≡ Skolemisierung

$(\forall x \in Z: (\forall i \geq 0: x \in g_M^i(\varnothing) \wedge x \in \overline{B} \Rightarrow x \in M))$

$\equiv$ $\forall$-split

$(\forall x \in Z: (x \in g_M^0(\varnothing) \wedge x \in \overline{B} \Rightarrow x \in M) \wedge (\forall i \geq 1: x \in g_M^i(\varnothing) \wedge x \in \overline{B} \Rightarrow x \in M))$

$\equiv$

$(\forall x \in Z: (x \in \varnothing \wedge x \in \overline{B} \Rightarrow x \in M) \wedge (\forall i \geq 1: x \in g_M^i(\varnothing) \wedge x \in \overline{B} \Rightarrow x \in M))$

$\equiv$ Logik

$(\forall x \in Z: (\mathit{false} \wedge x \in \overline{B} \Rightarrow x \in M) \wedge (\forall i \geq 1: x \in g_M^i(\varnothing) \wedge x \in \overline{B} \Rightarrow x \in M))$

$\equiv$ Logik

$(\forall x \in Z: (\forall i \geq 1: x \in g_M^i(\varnothing) \wedge x \in \overline{B} \Rightarrow x \in M))$

$\equiv$

$(\forall x \in Z: (\forall i \geq 0: x \in g_M(g_M^i(\varnothing)) \wedge x \in \overline{B} \Rightarrow x \in M))$

$\equiv$ Definition g_M

$(\forall x \in Z: (\forall i \geq 0: x \in \overline{B} \cap M \cup B \cap LP(P, g_M^i(\varnothing)) \wedge x \in \overline{B} \Rightarrow x \in M))$

$\equiv$ Mengenlehre

$(\forall x \in Z: (\forall i \geq 0: (x \in \overline{B} \wedge x \in M \vee x \in B \cap LP(P, g_M^i(\varnothing))) \wedge x \in \overline{B} \Rightarrow x \in M))$

$\equiv$ Logik

$(\forall x \in Z: (\forall i \geq 0: x \in \overline{B} \wedge x \in M \vee x \in B \cap LP(P, g_M^i(\varnothing)) \wedge x \in \overline{B} \Rightarrow x \in M))$

$\equiv$ Logik

$(\forall x \in Z: (\forall i \geq 0: (x \in \overline{B} \wedge x \in M \vee \mathit{false}) \Rightarrow x \in M))$

$\equiv$ Logik

true

•

Lemma 3.23: $Z \subseteq Z_k \equiv wp(P, Z \subseteq Z_{k-1})$.

Beweis:

$Z \subseteq Z_k \equiv wp(P, Z \subseteq Z_{k-1})$

$\equiv$ Mengenlehre

$(\forall t \in Z: t \in Z_k) \equiv wp(P, (\forall t \in Z: t \in Z_{k-1}))$

$\equiv$ Konjunktivität

$(\forall t \in Z: t \in Z_k) \equiv (\forall t \in Z: wp(P, t \in Z_{k-1}))$

$\Leftarrow$ Logik

$(\forall t \in Z: t \in Z_k \equiv wp(P, t \in Z_{k-1}))$

$\equiv$ Mengenlehre, Definition von *wp*

$(\forall t \in Z: C^{-1}(t \in Z_k) = LP(P, C^{-1}(t \in Z_{k-1})))$

≡ Mengenlehre, Definition von *LP*

$Z_k = \{s \in Z: f_P(s) \subseteq Z_{k-1}\}$

≡ Mengenlehre

$(\forall t \in Z: t \in Z_k \equiv t \in Z \wedge f_P(t) \subseteq Z_{k-1})$

≡ Lemma 3.15

true

•

Lemma 3.24: $[b \Rightarrow \neg(Z \subseteq Z_0)] \wedge [b \vee \neg N \equiv \neg(Z \subseteq Z_0)]$.

Beweis: (das 1. Konjunkt wird durch die folgende Implikationskette gezeigt; das 2. Konjunkt wird durch die Äquivalenzkette gezeigt, die in der 3. Zeile des Beweises beginnt)

b

$\Rightarrow$ Logik

$b \vee \neg N$

≡ De Morgan

$\neg(\neg b \wedge N)$

≡ Logik

$\neg(true \Rightarrow \neg b \wedge N)$

≡ Isomorphie

$\neg(C^{-1}(true) \subseteq C^{-1}(\neg b \wedge N))$

≡ Isomorphie

$\neg(Z \subseteq \overline{B} \cap M)$

≡ Definition von Z_0

$\neg(Z \subseteq Z_0)$

•

Lemma 3.25: $[V \Rightarrow wp(LOOP, N)] \Rightarrow (\exists k \geq 0 : Z \subseteq Z_k)$.

Beweis:

$C^{-1}(V) \subseteq LP(LOOP, M) \Rightarrow (\exists k \geq 0 : Z \subseteq Z_k)$

≡ While-Regel

$C^{-1}(V) \subseteq \bigcup_{i \geq 0} g_M^i(\varnothing) \Rightarrow (\exists k \geq 0 : Z \subseteq Z_k)$

≡ Mengenlehre, Lemma 3.11

$(\exists i \geq 0 : C^{-1}(V) \subseteq g_M^i(\varnothing)) \Rightarrow (\exists k \geq 0 : Z \subseteq Z_k)$

≡ Logik

$(\forall i \geq 0 : C^{-1}(V) \subseteq g_M^i(\varnothing) \Rightarrow (\exists k \geq 0 : Z \subseteq Z_k))$

Das wird durch Induktion über i bewiesen:

$i = 0$ (unter Verwendung von Zustandsmengen):

$C^{-1}(V) \subseteq g_M^0(\varnothing)$

$\equiv$

$C^{-1}(V) \subseteq \varnothing$

$\equiv$ Voraussetzung $C^{-1}(V) \neq \varnothing$

false

$\Rightarrow$ Logik

$(\exists k \geq 0 : Z \subseteq Z_k)$

$i \to i+1$ (unter Verwendung von Prädikaten): Voraussetzung

$[V \Rightarrow h_N^i(\mathit{false})] \Rightarrow (\exists k \geq 0 : Z \subseteq Z_k)$

$\equiv$ Everywhere-Operator, Logik

$(\forall \mathit{free}(V) \cup \mathit{free}(h_N^i(\mathit{false})) : \neg V \vee h_N^i(\mathit{false})) \Rightarrow (\exists k \geq 0 : Z \subseteq Z_k)$

$\equiv$ Logik

$(\exists \mathit{free}(V) \cup \mathit{free}(h_N^i(\mathit{false})) : \neg V \vee h_N^i(\mathit{false}) \Rightarrow (\exists k \geq 0 : Z \subseteq Z_k))$

$\equiv$ Logik

$$(\exists \mathit{free}(V) \cup \mathit{free}(h_N^i(\mathit{false})) : (\neg V \Rightarrow (\exists k \geq 0 : Z \subseteq Z_k)) \wedge (h_N^i(\mathit{false}) \Rightarrow (\exists k \geq 0 : Z \subseteq Z_k)))$$

zu zeigen ist

$$(\exists \mathit{free}(V) \cup \mathit{free}(h_N^{i+1}(\mathit{false})) : (\neg V \Rightarrow (\exists k \geq 0 : Z \subseteq Z_k)) \wedge (h_N^{i+1}(\mathit{false}) \Rightarrow (\exists k \geq 0 : Z \subseteq Z_k)))$$

Also

$$\begin{aligned}&(\exists \mathit{free}(V) \cup \mathit{free}(h_N^i(\mathit{false})) : (\neg V \Rightarrow (\exists k \geq 0 : Z \subseteq Z_k)) \wedge (h_N^i(\mathit{false}) \Rightarrow (\exists k \geq 0 : Z \subseteq Z_k)))\\ \Rightarrow\ &(\exists \mathit{free}(V) \cup \mathit{free}(h_N^{i+1}(\mathit{false})) : (\neg V \Rightarrow (\exists k \geq 0 : Z \subseteq Z_k)) \wedge (h_N^{i+1}(\mathit{false}) \Rightarrow (\exists k \geq 0 : Z \subseteq Z_k)))\end{aligned}$$

$\equiv$ Skolemisierung, gebundene Umbenennung

$$\begin{aligned}&(\forall \mathit{free}(V) \cup \mathit{free}(h_N^i(\mathit{false})) : (\neg V \Rightarrow (\exists k \geq 0 : Z \subseteq Z_k)) \wedge (h_N^i(\mathit{false}) \Rightarrow (\exists k \geq 0 : Z \subseteq Z_k))\\ &\quad \Rightarrow (\exists \mathit{free'}(V) \cup \mathit{free'}(h_N^{i+1}(\mathit{false})) : (\neg V \Rightarrow (\exists k \geq 0 : Z \subseteq Z_k)) \wedge (h_N^{i+1}(\mathit{false}) \Rightarrow (\exists k \geq 0 : Z \subseteq Z_k))))\end{aligned}$$

$\Leftarrow$ Logik, Everywhere-Operator

$$\begin{aligned}&[(\neg V \Rightarrow (\exists k \geq 0 : Z \subseteq Z_k)) \wedge (h_N^i(\mathit{false}) \Rightarrow (\exists k \geq 0 : Z \subseteq Z_k))\\ &\quad \Rightarrow (\neg V \Rightarrow (\exists k \geq 0 : Z \subseteq Z_k)) \wedge (h_N^{i+1}(\mathit{false}) \Rightarrow (\exists k \geq 0 : Z \subseteq Z_k))]\end{aligned}$$

$\equiv$ Logik

$$[(\neg V \Rightarrow (\exists k \geq 0 : Z \subseteq Z_k)) \wedge (h_N^i(\mathit{false}) \Rightarrow (\exists k \geq 0 : Z \subseteq Z_k))$$
$$\Rightarrow (h_N^{i+1}(\mathit{false}) \Rightarrow (\exists k \geq 0 : Z \subseteq Z_k))]$$

$\Leftarrow$ Logik

$$[(h_N^i(\mathit{false}) \Rightarrow (\exists k \geq 0 : Z \subseteq Z_k)) \Rightarrow (h_N^{i+1}(\mathit{false}) \Rightarrow (\exists k \geq 0 : Z \subseteq Z_k))]$$

Jetzt ist $h_N^{i+1}(\mathit{false}) \Rightarrow (\exists k \geq 0 : Z \subseteq Z_k)$ zu zeigen unter der Voraussetzung $h_N^i(\mathit{false}) \Rightarrow (\exists k \geq 0 : Z \subseteq Z_k)$:

$h_N^{i+1}(\mathit{false})$

$\equiv$

$h_N(h_N^i(\mathit{false}))$

$\equiv$ Definition von h_N

$\neg b \wedge N \vee b \wedge wp(P, h_N^i(\mathit{false}))$

$\Rightarrow$ Voraussetzung, Monotonie

$\neg b \wedge N \vee b \wedge wp(P, (\exists k \geq 0 : Z \subseteq Z_k))$

$\equiv$ Starke Disjunktivität, $\{Z_k\}$ ist eine endliche Kette

$\neg b \wedge N \vee b \wedge (\exists k \geq 0 : wp(P, Z \subseteq Z_k))$

$\equiv$ Lemma 3.23

$\neg b \wedge N \vee b \wedge (\exists k \geq 0 : Z \subseteq Z_{k+1})$

$\Rightarrow$ Lemma 3.24 (2. Konjunkt negiert, 1. Konjunkt), Indexverschiebung

$Z \subseteq Z_0 \vee \neg Z \subseteq Z_0 \wedge (\exists k \geq 1 : Z \subseteq Z_k)$

$\equiv$ Logik

$Z \subseteq Z_0 \vee (\exists k \geq 1 : Z \subseteq Z_k)$

$\equiv$ $\exists$-split

$(\exists k \geq 0 : Z \subseteq Z_k)$

•

Satz 3.26 (While-Korrektheit):

$[V \Rightarrow wp(LOOP, N)] \Rightarrow$
$(\exists J \in Rp(PS), t \in \mathbf{N}^W : [V \Rightarrow J] \wedge [J \wedge \neg b \Rightarrow N] \wedge$
$[J \wedge b \Rightarrow wp(P, J)] \wedge [J \wedge b \Rightarrow t \geq 0] \wedge [J \wedge b \wedge t = T \Rightarrow wp(P, t < T)])$

Beweis:

$[V \Rightarrow wp(LOOP, N)] \Rightarrow$
$(\exists J \in Rp(PS), t \in \mathbf{N}^W : [V \Rightarrow J] \wedge [J \wedge \neg b \Rightarrow N] \wedge$
$[J \wedge b \Rightarrow wp(P, J)] \wedge [J \wedge b \Rightarrow t \geq 0] \wedge [J \wedge b \wedge t = T \Rightarrow wp(P, t < T)])$

$\equiv$ mit $J \equiv wp(LOOP, N)$

$$[V \Rightarrow wp(LOOP, N)] \Rightarrow$$
$$(\exists t \in \mathbf{N}^W : [V \Rightarrow wp(LOOP, N)] \wedge [wp(LOOP, N) \wedge \neg b \Rightarrow N] \wedge$$
$$[wp(LOOP, N) \wedge b \Rightarrow wp(P, wp(LOOP, N))] \wedge$$
$$[wp(LOOP, N) \wedge b \Rightarrow t \geq 0] \wedge [wp(LOOP, N) \wedge b \wedge t = T \Rightarrow wp(P, t < T)])$$

$\equiv$ Logik

$$[V \Rightarrow wp(LOOP, N)] \Rightarrow$$
$$[V \Rightarrow wp(LOOP, N)] \wedge [wp(LOOP, N) \wedge \neg b \Rightarrow N] \wedge$$
$$[wp(LOOP, N) \wedge b \Rightarrow wp(P, wp(LOOP, N))] \wedge$$
$$(\exists t \in \mathbf{N}^W : [wp(LOOP, N) \wedge b \Rightarrow t \geq 0] \wedge [wp(LOOP, N) \wedge b \wedge t = T \Rightarrow wp(P, t < T)])$$

$\equiv$ Logik

$$[V \Rightarrow wp(LOOP, N)] \Rightarrow$$
$$[wp(LOOP, N) \wedge \neg b \Rightarrow N] \wedge$$
$$[wp(LOOP, N) \wedge b \Rightarrow wp(P, wp(LOOP, N))] \wedge$$
$$(\exists t \in \mathbf{N}^W : [wp(LOOP, N) \wedge b \Rightarrow t \geq 0] \wedge [wp(LOOP, N) \wedge b \wedge t = T \Rightarrow wp(P, t < T)])$$

$\equiv$ mit $wp(LOOP, N)$ ist innerhalb der Schleife invariant [F93]

$$[V \Rightarrow wp(LOOP, N)] \Rightarrow$$
$$[wp(LOOP, N) \wedge \neg b \Rightarrow N] \wedge$$
$$(\exists t \in \mathbf{N}^W : [wp(LOOP, N) \wedge b \Rightarrow t \geq 0] \wedge [wp(LOOP, N) \wedge b \wedge t = T \Rightarrow wp(P, t < T)])$$

$\equiv$ Lemma 3.22

$$[V \Rightarrow wp(LOOP, N)] \Rightarrow$$
$$(\exists t \in \mathbf{N}^W : [wp(LOOP, N) \wedge b \Rightarrow t \geq 0] \wedge [wp(LOOP, N) \wedge b \wedge t = T \Rightarrow wp(P, t < T)])$$

$\equiv$ mit $t = \min\{k \geq 0 : Z \subseteq Z_k\}$, was wegen der Voraussetzung $[V \Rightarrow wp(LOOP, N)]$ und Lemma 3.25 existiert

$$[V \Rightarrow wp(LOOP, N)] \Rightarrow [wp(LOOP, N) \wedge b \Rightarrow \min\{k \geq 0 : Z \subseteq Z_k\} \geq 0] \wedge$$
$$[wp(LOOP, N) \wedge b \wedge \min\{k \geq 0 : Z \subseteq Z_k\} = T \Rightarrow wp(P, \min\{k \geq 0 : Z \subseteq Z_k\} < T)]$$

$\equiv$ Definition von min

$$[V \Rightarrow wp(LOOP, N)] \Rightarrow [wp(LOOP, N) \wedge b \Rightarrow true] \wedge$$
$$[wp(LOOP, N) \wedge b \wedge (\forall U : 0 \leq T \leq U : Z \subseteq Z_U) \wedge Z \subseteq Z_T \Rightarrow wp(P, (\exists k : 0 \leq k < T : Z \subseteq Z_k))]$$

$\Leftarrow$ Logik, Schwache Disjunktivität

$$[V \Rightarrow wp(LOOP, N)] \Rightarrow$$
$$[wp(LOOP, N) \wedge b \wedge (\forall U : 0 \leq T \leq U : Z \subseteq Z_U) \wedge Z \subseteq Z_T \Rightarrow (\exists k : 0 \leq k < T : wp(P, Z \subseteq Z_k))]$$

$\Leftarrow$ Lemma 3.24 (1. Konjunkt)

$$[V \Rightarrow wp(LOOP, N)] \Rightarrow$$
$$[wp(LOOP, N) \wedge b \wedge (\forall U : 0 \leq T \leq U : Z \subseteq Z_U) \wedge \neg(Z \subseteq Z_0) \wedge Z \subseteq Z_T \Rightarrow (\exists k : 0 \leq k < T : wp(P, Z \subseteq Z_k))]$$

$\Leftarrow$ Logik

$[V \Rightarrow wp(LOOP, N)] \Rightarrow$

$[wp(LOOP, N) \wedge b \wedge (\forall U{:}0 \le T \le U{:} Z \subseteq Z_U) \wedge Z \subseteq Z_T \wedge T \ge 1 \Rightarrow (\exists k{:}0 \le k < T{:} wp(P, Z \subseteq Z_k))]$

$\equiv$ mit $k = T - 1$

$[V \Rightarrow wp(LOOP, N)] \Rightarrow$

$[wp(LOOP, N) \wedge b \wedge (\forall U{:}0 \le T \le U{:} Z \subseteq Z_U) \wedge Z \subseteq Z_T \wedge T \ge 1 \Rightarrow 0 \le T - 1 < T \wedge wp(P, Z \subseteq Z_{T-1})]$

$\equiv$ Logik

$[V \Rightarrow wp(LOOP, N)] \Rightarrow$

$[wp(LOOP, N) \wedge b \wedge (\forall U{:}0 \le T \le U{:} Z \subseteq Z_U) \wedge Z \subseteq Z_T \wedge T \ge 1 \Rightarrow wp(P, Z \subseteq Z_{T-1})]$

$\equiv$ Lemma 3.23

$[V \Rightarrow wp(LOOP, N)] \Rightarrow$

$[wp(LOOP, N) \wedge b \wedge (\forall U{:}0 \le T \le U{:} Z \subseteq Z_U) \wedge wp(P, Z \subseteq Z_{T-1}) \wedge T \ge 1 \Rightarrow wp(P, Z \subseteq Z_{T-1})]$

$\equiv$ Logik

true

•

Bemerkung: Die Sätze 3.21 und 3.26 bedeuten, daß die Existenz von Invariante und Terminierungsfunktion äquivalent zur Korrektheit der Schleife sind. Damit ist die *Existenz* von Invariante und Terminierungsfunktion eine exakte VC. Doch bei der Verifikation von Schleifen geht man von einer konkreten Invarianten und Terminierungsfunktion aus, und die damit gebildeten VCs sind approximierte VCs. Das kann man auch formal zeigen. Dazu sei P ein zweistelliges Prädikat über Prädikaten 1. Ordnung und Funktionen. $P(I,t)$ bedeutet, daß I Invariante und t Terminierungsfunktion einer Schleife sind, d.h. die 5 Bedingungen von Satz 3.21 erfüllen. K bezeichne die Korrektheit der Schleife (die Vorbedingung der Schleife soll nicht konstant *false* sein). Damit lautet die Konjunktion der Sätze 3.21 und 3.26: $(\exists X, f : P(X, f)) \equiv K$. Die konkrete, gegebene Invariante und Terminierungsfunktion der Schleife seien I und t, die VC lautet damit $P(I,t)$, d.h. die Gültigkeit der VC impliziert die Korrektheit $P(I,t) \Rightarrow K$. Damit ist aber die VC noch keine exakte VC, d.h. es gilt nicht unbedingt $K \Rightarrow P(I,t)$.

Beweis durch Widerspruch: angenommen, unter den Voraussetzungen $(\exists X, f : P(X, f)) \equiv K$ und $P(I,t) \Rightarrow K$ gelte $K \Rightarrow P(I,t)$, d.h.

$((\exists X, f : P(X, f)) \equiv K) \wedge (P(I,t) \Rightarrow K) \Rightarrow (K \Rightarrow P(I,t))$

$\equiv$ Substitution

$(P(I,t) \Rightarrow (\exists X, f : P(X, f))) \Rightarrow ((\exists X, f : P(X, f)) \Rightarrow P(I,t))$

$\equiv$ Logik

$true \Rightarrow ((\exists X, f : P(X, f)) \Rightarrow P(I,t))$

•

Das gilt nicht unbedingt. Damit ist verdeutlicht, daß die 5 Konjunkte in der Voraussetzung von Satz 3.21 eine approximierte und im Allgemeinen keine exakte VC sind.

3.4.2 For-Schleifen

For-Schleifen kann man als spezielle While-Schleifen betrachten. Es stellt sich daher die Frage, warum man überhaupt eine Beweisregel für For-Schleifen herleiten sollte.

1. In Ada und anderen imperativen / prozeduralen Programmiersprachen gibt es im Allgemeinen For-Schleifen. Dann könnte man aber beim Beweis von For-Schleifen intern eine Transformation in eine While-Schleife vornehmen und mit der Beweisregel für While-Schleifen die For-Schleife beweisen.
2. Damit nutzt man aber nicht die speziellen Eigenschaften der For-Schleife aus. Das ist etwa analog zu einem numerischen Verfahren zur Lösung linearer Gleichungssysteme, die ein Spezialfall von nicht linearen Gleichungssystemen sind. Zur Lösung eines linearen Gleichungssystems kann man ein Verfahren zur Lösung eines nicht linearen Gleichungssystems verwenden, wobei man die Linearität „verschenkt". Ein spezielles Verfahren zur Lösung eines linearen Gleichungssystems ergibt sich auch nicht automatisch durch Modifikation des Verfahrens zur Lösung eines nicht linearen Gleichungssystems, sondern ist ein eigenes Verfahren. Durch Ausnutzung der einfacheren Struktur linearer Gleichungen vereinfacht sich das Verfahren und Sätze sowie Beweise dieser Sätze über das Verfahren. Genauso ist es auch hier.

Es gibt eine Beweisregel von Hoare [H72], die in der Literatur nicht weiter verfolgt wurde und einige Nachteile hat:

1. keine Berücksichtigung der leeren Schleife
2. Die Invariante ist ein Prädikat über einem Intervall $P([a \ldots i])$ vom ersten Wert a der Schleifenvariablen bis zum aktuellen Wert i. Vor Ausführung der Schleife gilt das leere Intervall $P([])$. Dabei ist nicht klar, wie das entsprechende Prädikat lautet. Man könnte z.B. $P([a \ldots pred(a)])$ nehmen, was oft funktioniert, wenn man mit ganzen Zahlen zu tun hat. Dabei kann es aber Schwierigkeiten bei Aufzählungstypen geben, wenn $pred(a)$ nicht definiert ist.
3. Die Invariante ist wegen der Anschlußbedingung am Anfang schwer zu finden, wie das folgende Beispiel zeigt [W98]:

```
-- V: x = 5
for i in 1 .. 10 loop
   x := i;
   -- P: ???
end loop;
-- N: x = 10;
```

4. In der Invarianten darf nicht die Schleifenvariable vorkommen.

Eine einfache und offensichtliche Invariante für das Beispiel in Punkt 3 wäre $x = i$, was wegen Punkt 4 nicht zulässig ist. Außerdem paßt damit die Anschlußbedingung am Anfang nicht. Alle 4 Nachteile werden durch den folgenden Ansatz beseitigt.

3.4.2.1 Aufwärtszählende For-Schleifen

Eine aufwärtszählende For-Schleife hat die Form

```
-- V
for I in A .. B loop
   S;
   -- P
end loop;
-- N
```

Dabei ist *I* die Schleifenvariable, *A* und *B* die Schleifengrenzen, *V* und *N* sind Vor- bzw. Nachbedingung und *P* ist die Invariante. Weder *I* noch *A* und *B* werden im Schleifenrumpf *S* geändert, und es wird vorausgesetzt, daß *S* terminiert und bei Auswertung von *A* und *B* sowie *S* keine Seiteneffekte auftreten. Im Allgemeinen kommt *I* in *P* und/oder *S* vor, d.h. es wird lesend auf *I* zugegriffen. Außerdem ist *I* nach Ende der Schleife nicht mehr sichtbar. Das bedeutet insbesondere, daß *I* in der Nachbedingung *N* nicht vorkommen darf.

Bemerkung: *A* und *B* sind Ausdrücke eines diskreten Typs, auf dem die Funktionen *pred* (Vorgänger, außer am unteren Rand) und *succ* (Nachfolger, außer am oberen Rand) definiert sind. Auf einem Integer-Typ entsprechen diese Funktionen +1 und -1. Im Folgenden wird für $pred(A)$ und $succ(A)$ $A+1$ bzw. $A-1$ geschrieben, um die Formeln kürzer zu halten. In diesem Abschnitt ist mit For-Schleife immer eine aufwärtszählende For-Schleife gemeint.

•

Die Semantik der For-Schleife ist [TD95: 112]:
Die Schleifenvariable bekommt den ersten Wert des zu durchlaufenden Bereiches zugewiesen. Für jeden Wert des Bereiches wird der Schleifenrumpf *S* ausgeführt. Nach jeder Ausführung von *S* wird der Schleifenvariablen der nächste Wert des Bereiches zugewiesen. Nach Ausführung der Schleife ist *I* nicht mehr sichtbar. Falls der zu durchlaufenden Bereich leer ist, wird der Schleifenrumpf gar nicht ausgeführt. Damit ist [W98]

$$FOR \equiv \begin{cases} S_A^I; S_{A+1}^I; S_{A+1}^I; \ldots; S_{B-1}^I; S_B^I;, & falls\ A \le B \\ NULL, & falls\ A > B \end{cases}$$

Mit den Zusicherungen sieht die aufgerollte Schleife dann so aus:

```
-- V
S^I_A;
-- P^I_A
S^I_A+1;
-- P^I_A+1
S^I_A+2;
-- P^I_A+2
     :
     :
S^I_B-1;
-- P^I_B-1
S^I_B;
-- P^I_B
-- N
```

Daraus kann man die folgenden Implikationen für den Fall $A \le B$ ablesen:

$[V \Rightarrow wp(S_A^I, P_A^I)]$, $[P_A^I \Rightarrow wp(S_{A+1}^I, P_{A+1}^I)]$, $[P_{A+2}^I \Rightarrow wp(S_{A+2}^I, P_{A+2}^I)]$, ..., $[P_{B-1}^I \Rightarrow wp(S_B^I, P_B^I)]$, $[P_B^I \Rightarrow N]$. Für $A > B$ gilt $[V \Rightarrow N]$. Damit erhält man folgende Beweisregel:

$$[A \le B \Rightarrow (V \Rightarrow wp(S_A^I, P_A^I)] \wedge [(\forall j : A+1 \le j \le B : P_{j-1}^I \Rightarrow wp(S_j^I, P_j^I))] \wedge [(P_B^I \Rightarrow N)] \wedge$$
$$[A > B \Rightarrow (V \Rightarrow N)]$$

Das kann äquivalent umgeformt werden in die folgende Konjunktion mit 4 Konjunkten:

$$[(A \le B \wedge V \Rightarrow wp(S_A^I, P_A^I)) \wedge (A \le B \wedge (\forall j : A+1 \le j \le B : P_{j-1}^I \Rightarrow wp(S_j^I, P_j^I))) \wedge$$
$$(A \le B \wedge P_B^I \Rightarrow N) \wedge (A > B \wedge V \Rightarrow N)]$$

Im 2. Konjunkt kann man A ≤ B weglassen und erhält

Satz 3.27 (For-Regel):

$$[(A \le B \wedge V \Rightarrow wp(S_A^I, P_A^I)) \wedge (\forall j : A+1 \le j \le B : P_{j-1}^I \Rightarrow wp(S_j^I, P_j^I)) \wedge$$
$$(A \le B \wedge P_B^I \Rightarrow N) \wedge (A > B \wedge V) \Rightarrow N]$$

ist eine VC für For-Schleifen. (Auf die 4 Konjunkte wird in den Beispielen mit (1), (2), (3) und (4) verwiesen.)
Beweis: zu zeigen ist, daß die For-Regel die Korrektheit der For-Schleife impliziert, also

$$[(A \le B \wedge V \Rightarrow wp(S_A^I, P_A^I)) \wedge (\forall j: A+1 \le j \le B: P_{j-1}^I \Rightarrow wp(S_j^I, P_j^I)) \wedge$$
$$(A \le B \wedge P_B^I \Rightarrow N) \wedge (A > B \wedge V \Rightarrow N)] \Rightarrow [V \Rightarrow wp(FOR, N)]$$

$$[(A \le B \wedge V \Rightarrow wp(S_A^I, P_A^I)) \wedge (\forall j: A+1 \le j \le B: P_{j-1}^I \Rightarrow wp(S_j^I, P_j^I)) \wedge$$
$$(A \le B \wedge P_B^I \Rightarrow N) \wedge (A > B \wedge V \Rightarrow N)] \Rightarrow [V \Rightarrow wp(FOR, N)]$$

≡ Logik

$$[A \le B \wedge (V \Rightarrow wp(S_A^I, P_A^I)) \wedge (\forall j: A+1 \le j \le B: P_{j-1}^I \Rightarrow wp(S_j^I, P_j^I)) \wedge (P_B^I \Rightarrow N)] \vee$$
$$[A > B \wedge (V \Rightarrow N)] \Rightarrow [V \Rightarrow wp(FOR, N)]$$

≡ Logik, Definition FOR

$$([A \le B \wedge (V \Rightarrow wp(S_A^I, P_A^I)) \wedge (\forall j: A+1 \le j \le B: P_{j-1}^I \Rightarrow wp(S_j^I, P_j^I)) \wedge (P_B^I \Rightarrow N)] \Rightarrow$$
$$[V \Rightarrow wp(S_A^I; S_{A+1}^I; S_{A+2}^I; \ldots; S_{B-1}^I; S_B^I, N)]) \wedge$$
$$[A > B \wedge (V \Rightarrow N)] \Rightarrow [V \Rightarrow wp(NULL, N)])$$

≡ Null-Regel, Logik

$$[A \le B \wedge (V \Rightarrow wp(S_A^I, P_A^I)) \wedge (\forall j: A+1 \le j \le B: P_{j-1}^I \Rightarrow wp(S_j^I, P_j^I)) \wedge (P_B^I \Rightarrow N)] \Rightarrow$$
$$[V \Rightarrow wp(S_A^I; S_{A+1}^I; S_{A+2}^I; \ldots; S_{B-1}^I; S_B^I, N)]$$

⇐ Logik

$$[A \le B \wedge (V \Rightarrow wp(S_A^I, P_A^I)) \wedge (\forall j: A+1 \le j \le B: P_{j-1}^I \Rightarrow wp(S_j^I, P_j^I)) \wedge (P_B^I \Rightarrow N) \Rightarrow$$
$$(V \Rightarrow wp(S_A^I; S_{A+1}^I; S_{A+2}^I; \ldots; S_{B-1}^I; S_B^I, N))]$$

≡ Portation

$$[A \le B \wedge V \wedge (V \Rightarrow wp(S_A^I, P_A^I)) \wedge (\forall j: A+1 \le j \le B: P_{j-1}^I \Rightarrow wp(S_j^I, P_j^I)) \wedge (P_B^I \Rightarrow N) \Rightarrow$$
$$wp(S_A^I; S_{A+1}^I; S_{A+2}^I; \ldots; S_{B-1}^I; S_B^I, N)]$$

⇐ Monotonie, Ersetzung von N durch P_B^I nach Konjunkt 5 der Voraussetzung

$$[A \le B \wedge V \wedge (V \Rightarrow wp(S_A^I, P_A^I)) \wedge (\forall j: A+1 \le j \le B: P_{j-1}^I \Rightarrow wp(S_j^I, P_j^I)) \wedge (P_B^I \Rightarrow N) \Rightarrow$$
$$wp(S_A^I; S_{A+1}^I; S_{A+2}^I; \ldots; S_{B-1}^I; S_B^I, P_B^I)]$$

≡ Sequenzregel

$$[A \le B \wedge V \wedge (V \Rightarrow wp(S_A^I, P_A^I)) \wedge (\forall j: A+1 \le j \le B: P_{j-1}^I \Rightarrow wp(S_j^I, P_j^I)) \wedge (P_B^I \Rightarrow N) \Rightarrow$$
$$wp(S_A^I, wp(S_{A+1}^I, wp(S_{A+2}^I, \ldots wp(S_{B-1}^I, wp(S_B^I, P_B^I) \cdots)]$$

$\Leftarrow$ Monotonie, Ersetzung von $wp(S_B^I, P_B^I)$ durch P_{B-1}^I nach Konjunkt 4 der Voraussetzung

$[A \le B \wedge V \wedge (V \Rightarrow wp(S_A^I, P_A^I)) \wedge (\forall j: A+1 \le j \le B: P_{j-1}^I \Rightarrow wp(S_j^I, P_j^I)) \wedge (P_B^I \Rightarrow N) \Rightarrow$
$wp(S_A^I, wp(S_{A+1}^I, wp(S_{A+2}^I, \ldots wp(S_{B-1}^I, P_{B-1}^I)\cdots)]$

:

:

$\Leftarrow$ Monotonie, Ersetzung von $wp(S_{A+1}^I, P_{A+1}^I)$ durch P_A^I nach Konjunkt 4 der Voraussetzung

$[A \le B \wedge V \wedge (V \Rightarrow wp(S_A^I, P_A^I)) \wedge (\forall j: A+1 \le j \le B: P_{j-1}^I \Rightarrow wp(S_j^I, P_j^I)) \wedge (P_B^I \Rightarrow N) \Rightarrow wp(S_A^I, P_A^I)]$
$\equiv$ modus ponens (von Konjunkt 2 mit Konjunkt 3 der Voraussetzung)

true

•

3.4.2.2 Existenz der Invarianten

Die For-Regel besagt, daß die Existenz eines Prädikates *P*, das die Voraussetzungen von Satz 3.27 erfüllt (einer Invarianten also), die Korrektheit einer For-Schleife impliziert. Nun stellt sich die Frage, ob auch die Umkehrung gilt, d.h. ob jede korrekte For-Schleife eine Invariante *P* hat. Daß dies gilt, wird im Folgenden gezeigt. Das folgt aber auch schon daraus, daß dieselbe Aussage für die allgemeineren While-Schleifen zutrifft. Dabei wird von der Korrektheit ausgegangen und gezeigt, welches Prädikat eine Invariante ist. Das ist allerdings kein konstruktiver Beweis in dem Sinne, daß man mit dessen Hilfe eine Invariante mechanisch konstruieren könnte.

Satz 3.28: Zu einer korrekten For-Schleife existiert eine Invariante.
Beweis: zu zeigen ist

$[V \Rightarrow wp(FOR, N)] \Rightarrow (\exists P: [(A \le B \wedge V \Rightarrow wp(S_A^I, P_A^I)) \wedge (\forall j: A+1 \le j \le B: P_{j-1}^I \Rightarrow wp(S_j^I, P_j^I)) \wedge$
$(A \le B \wedge P_B^I \Rightarrow N) \wedge (A > B \wedge V \Rightarrow N)]$

$[V \Rightarrow wp(FOR, N)] \Rightarrow (\exists P: [(A \le B \wedge V \Rightarrow wp(S_A^I, P_A^I)) \wedge (\forall j: A+1 \le j \le B: P_{j-1}^I \Rightarrow wp(S_j^I, P_j^I)) \wedge$
$(A \le B \wedge P_B^I \Rightarrow N) \wedge (A > B \wedge V \Rightarrow N)]$

$\equiv$ mit $P \equiv wp(\textit{ for } K \textit{ in } I+1..B \textit{ loop } S_K^I \textit{ end loop}, N)$

$[V \Rightarrow wp(FOR, N)] \Rightarrow$
$[(A \le B \wedge V \Rightarrow wp(S_A^I, wp(\textit{ for } K \textit{ in } A+1..B \textit{ loop } S_K^I \textit{ end loop}, N))) \wedge$
$(\forall j: A+1 \le j \le B: wp(\textit{ for } K \textit{ in } j..B \textit{ loop } S_K^I \textit{ end loop}, N) \Rightarrow$
$wp(S_j^I, wp(\textit{ for } K \textit{ in } j+1..B \textit{ loop } S_K^I \textit{ end loop}, N))) \wedge$
$(A \le B \wedge wp(\textit{ for } K \textit{ in } B+1..B \textit{ loop } S_K^I \textit{ end loop}, N) \Rightarrow N) \wedge (A > B \wedge V \Rightarrow N)]$

$\equiv$ Sequenzregel, leere Schleife

$[V \Rightarrow wp(FOR, N)] \Rightarrow$
$[(A \le B \wedge V \Rightarrow wp(\textit{ for } K \textit{ in } A..B \textit{ loop } S_K^I \textit{ end loop}, N)) \wedge$
$(\forall j: A+1 \le j \le B: wp(\textit{ for } K \textit{ in } j..B \textit{ loop } S_K^I \textit{ end loop}, N) \Rightarrow$
$wp(\textit{ for } K \textit{ in } j..B \textit{ loop } S_K^I \textit{ end loop}, N)) \wedge$
$(A \le B \wedge wp(NULL, N) \Rightarrow N) \wedge (A > B \wedge V \Rightarrow N)]$

≡ Definition von FOR, Logik, Null-Regel

$[V \Rightarrow wp(FOR, N)] \Rightarrow$

$[(A \le B \wedge V \Rightarrow wp(FOR, N)) \wedge (\forall j: A+1 \le j \le B: true) \wedge (A \le B \wedge N \Rightarrow N) \wedge (A > B \wedge V \Rightarrow N)]$

≡ Logik

$[V \Rightarrow wp(FOR, N)] \Rightarrow [(A \le B \wedge V \Rightarrow wp(FOR, N)) \wedge true \wedge true \wedge (A > B \wedge V \Rightarrow N)]$

≡ Logik

$[V \Rightarrow wp(FOR, N)] \Rightarrow [A > B \wedge V \Rightarrow N]$

⇐ Logik

$[(V \Rightarrow wp(FOR, N)) \Rightarrow (A > B \wedge V \Rightarrow N)]$

≡ portation

$[(V \Rightarrow wp(FOR, N)) \wedge A > B \wedge V \Rightarrow N]$

≡ modus ponens

$[wp(FOR, N) \wedge A > B \wedge V \Rightarrow N]$

≡ Definition FOR

$[N \wedge A > B \wedge V \Rightarrow N]$

≡ Logik

true

•

3.4.2.3 Abwärtszählende For-Schleifen

Auf analoge Weise erhält man eine VC für abwärtszählende For-Schleifen der Form

```
-- V
for I in reverse A .. B loop
   S;
   -- P
end loop;
-- N
```

Sie lautet

$$[(A \le B \wedge V \Rightarrow wp(S_B^I, P_B^I)) \wedge (\forall j: A \le j \le B-1: P_{j+1}^I \Rightarrow wp(S_j^I, P_j^I)) \wedge$$
$$(A \le B \wedge P_A^I \Rightarrow N) \wedge (A > B \wedge V \Rightarrow N)]$$

3.4.2.4 Beispiele

Es folgen einige Beispiele zur Illustration der Beweisregeln. Es werden nur die VCs (1) bis (4) und ihr Wahrheitswert angegeben. Auf eine ausführliche Umformung wird verzichtet.

Beispiel 1: Zunächst die Berechnung der Fakultät *f* einer Zahl *n* zwischen 0 und 20. *f* soll eine 64 bit Ganzzahl sein [KW97]. Durch die Typprüfung bei der Zuweisung wird bei der Berechnung von *wp* der Bereich von *f* automatisch als Konjunkt hinzugefügt.

```
-- V: 0 ≤ n ≤ 20 ∧ f = 1
for i in 1..n loop
   f := f * i;
   -- P: f = i! ∧ n ≤ 20
end loop;
-- N: f = n!
```

$$[1 \le n \wedge 0 \le n \le 20 \wedge f = 1 \Rightarrow wp(f := f * 1, f = 1! \wedge n \le 20)] \equiv true$$

$$wp(f := f * 1, f = 1! \wedge n \le 20)$$

$\equiv$ Zuweisungsregel

$$f \cdot 1 = 1! \wedge -2^{63} \le f \cdot 1 \le 2^{63} - 1 \wedge n \le 20$$

$$[2 \le i \le n \wedge f = (i-1)! \wedge n \le 20 \Rightarrow wp(f := f * i, f = i! \wedge n \le 20)] \equiv true$$

$$wp(f := f * i, f = i! \wedge n \le 20)$$

$\equiv$ Zuweisungsregel

$$f \cdot i = i! \wedge -2^{63} \le f \cdot i \le 2^{63} - 1 \wedge n \le 20$$

(3) $[1 \le n \le 20 \wedge f = n! \Rightarrow f = n!] \equiv true$

(4) $[1 > n \wedge 0 \le n \le 20 \wedge f = 1 \Rightarrow f = n!] \equiv true$

•

Beispiel 2: Wenn *n* als 32 bit Ganzzahl definiert wird und vor Ausführung der Schleife die Beschränkung $0 \le n \le 20$ nicht sichergestellt ist, ergibt sich das folgende falsche Programm zur Berechnung der Fakultät. Dieses Beispiel zeigt noch einmal die Wichtigkeit der Typprüfung [KW97].

```
-- V: -2^31 ≤ n ≤ 2^31-1 ∧ f = 1
for i in 1..n loop
   f := f * i;
   -- P: f = i! ∧ -2^31 ≤ n ≤ 2^31-1
end loop;
-- N: f = n!
```

$$[1 \le n \wedge -2^{31} \le n \le 2^{31} - 1 \wedge f = 1 \Rightarrow wp(f := f * 1, f = 1! \wedge -2^{31} \le n \le 2^{31} - 1)] \equiv true$$

$$wp(f := f * 1, f = 1! \wedge -2^{31} \le n \le 2^{31} - 1)$$

$\equiv$ Zuweisungsregel

$$f \cdot 1 = 1! \wedge -2^{63} \le f \cdot 1 \le 2^{63} - 1 \wedge -2^{31} \le n \le 2^{31} - 1$$

$$[2 \le i \le n \wedge f = (i-1)! \wedge -2^{31} \le n \le 2^{31} - 1 \Rightarrow wp(f := f * i, f = i! \wedge -2^{31} \le n \le 2^{31} - 1)]$$

$\equiv false$

$$wp(f := f * i, f \cdot i = i! \wedge -2^{31} \le n \le 2^{31} - 1)$$

$\equiv$ Zuweisungsregel

$$f \cdot i = i! \wedge -2^{63} \le f \cdot i \le 2^{63} - 1 \wedge -2^{31} \le n \le 2^{31} - 1$$

(3) $[1 \le n \wedge -2^{31} \le n \le 2^{31} - 1 \wedge f = n! \Rightarrow f = n!] \equiv true$

(4) $[1 > n \wedge -2^{31} \le n \le 2^{31} - 1 \wedge f = 1 \Rightarrow f = n!] \equiv false$

•

Beispiel 3: Bestimmung des Maximums *m* des nicht leeren Integer-Arrays *b*(1..*len*) durch eine abwärtszählende For-Schleife. Der Komponententyp von *b* liegt im Typ von *m*, und es kommen keine arithmetischen Operationen vor, so daß die Typprüfung immer *true* ergibt und im Folgenden weggelassen werden kann.

```
-- V: m = b(len) and len ≥ 1
for i in reverse 1 .. len loop
  if b(i) > m then m := b(i); end if;
  -- P: len ≥ 1 and (forall j: i ≤ j ≤ len: m ≥ b(j))
        and (exists j: i ≤ j ≤ len: m = b(j))
end loop;
-- N: (forall j: 1 ≤ j ≤ len: m ≥ b(j))
  and (exists j: 1 ≤ j ≤ len: m = b(j))
```

(1)

$$[1 \le len \wedge m = b(len) \wedge len \ge 1 \Rightarrow wp(\text{if } b(len) > m \text{ then } m := b(len); \text{ end if};,$$
$$len \ge 1 \wedge (\forall j : len \le j \le len : m \ge b(j)) \wedge (\exists j : len \le j \le len : m = b(j)))]$$

≡

true

(2)

$$[1 \le i \le len - 1 \wedge len \ge 1 \wedge (\forall j : i + 1 \le j \le len : m \ge b(j)) \wedge (\exists j : i + 1 \le j \le len : m = b(j)) \Rightarrow$$
$$b(i) > m \wedge len \ge 1 \wedge (\forall j : i \le j \le len : b(i) \ge b(j)) \wedge (\exists j : i \le j \le len : b(i) = b(j)) \vee$$
$$b(i) \le m \wedge len \ge 1 \wedge (\forall j : i \le j \le len : m \ge b(j)) \wedge (\exists j : i \le j \le len : m = b(j))]$$

≡

true

(3)

$$[1 \le len \wedge (\forall j : 1 \le j \le len : m \ge b(j)) \wedge (\exists j : 1 \le j \le len : m = b(j)) \Rightarrow$$
$$(\forall j : 1 \le j \le len : m \ge b(j)) \wedge (\exists j : 1 \le j \le len : m = b(j))]$$

≡ Logik

true

(4)

$$[1 > len \wedge m = b(len) \wedge 1 \le len \Rightarrow$$
$$(\forall j : 1 \le j \le len : m \ge b(j)) \wedge (\exists j : 1 \le j \le len : m = b(j))]$$

≡ Logik

true

3.4.3 Prozeduraufrufe

Dieser Abschnitt beinhaltet einen Vergleich von 4 in der Literatur vorkommenden Vorbedingungen für Prozeduraufrufe (Abschnitt 3.4.3.2). Der Vergleich zeigt, daß 3 dieser Vorbedingungen wirklich unterschiedlich sind, d.h. es gibt korrekte Aufrufe, die mit Hilfe einer Vorbedingung bewiesen werden können, mit Hilfe einer anderen Vorbedingungen jedoch nicht. Außerdem bilden die 4 Vorbedingungen eine Hierarchie bezüglich Leistungsfähigkeit. Das ist beim Betrachten der Formeln für die jeweiligen Vorbedingungen nicht sofort klar. Die leistungsfähigste, d.h. schwächste, dieser 4 Bedingungen, *vcb* [BMW89], wird verwendet, um leistungsfähige VCs für Schleifen im Kontext einer umgebenden Anweisung (Abschnitt 3.4.4) und rekursive Prozeduren zu erhalten (Abschnitt 3.4.5). Zudem wird die Anwendung von *vcb*

anhand einiger Beispiele erläutert. Zunächst wird gezeigt, wie mit bekannten Techniken Funktionen in Prozeduren und die entsprechenden Funktions- in semantisch äquivalente Prozeduraufrufe transformiert und globale Variablen eliminiert werden können (Abschnitt 3.4.3.1). Bei der Elimination globaler Variablen wird die semantische Äquivalenz der ursprünglichen und transformierten Programme formal mit Hilfe von Satz 2.31 gezeigt.
Es werden nur Aufrufe von Prozeduren ohne Seiteneffekte betrachtet. Seiteneffekte können durch in der Prozedur geänderte globale Variablen, Verweis-Typen als Parameter und den Parameterübergabemechanismus *call-by-reference* entstehen, sowie bei Auswertung eines aktuellen Parameters. Verweis-Typen werden in der vorliegenden Arbeit sowieso nicht betrachtet. Der Parameterübergabemechanismus für einfache Typen in Ada ist *copy in / copy out*. Für zusammengesetzte Typen kann implementierungsabhängig auch *call-by-reference* verwendet werden. Globale Variablen können immer eliminiert werden, was weiter unten noch gezeigt wird. Damit werden von den betrachteten Prozeduren nach außen sichtbar nur die out- und in-out-Parameter verändert.
Eine Prozedur ist spezifiziert durch

```
--! Pre: P;
--! Post: R;
procedure proc(x: in Typ; y: in out Typ; z: out Typ);
```

P ist die Vorbedingung, *R* die Nachbedingung, *x* die Liste der formalen in-Parameter, *y* die Liste der formalen in-out-Parameter, *z* die Liste der formalen out-Parameter und *proc* der Prozedurname. Die Listen können auch leer sein. *x* ist innerhalb der Prozedur eine (mit *X* initialisierte) lokale Konstante, *y* eine mit *Y* initialisierte, lokale Variable und *z* eine nicht unbedingt initialisierte, lokale Variable. *x* kann in *P* den Anfangswert *X* bekommen, *y* muß den Anfangswert *Y* haben. Die Vorbedingung darf keine Annahmen über den Wert von *z* machen, außer denen, die durch den Typ von *z* vorgegeben sind. *R* kann *y* und *z* in Abhängigkeit von *x* (bzw. *X*) und *Y* enthalten. Der Anfangswert *Y* von *y* muß gemerkt werden, da auf ihn in *R* Bezug genommen werden kann, *y* aber verändert werden kann. Die Variablen *X* und *Y* für die Anfangswerte sind keine Programmvariablen, sondern nur Variablen in den Zusicherungen, sogenannte *Spezifikationsvariablen*. Ihre Namen sind frei wählbar, solange sie eindeutig sind. Die Nachbedingung *R* kann immer so umgeformt werden, daß sie unabhängig von *x* ist, indem auf *X* Bezug genommen wird. Ein Prozeduraufruf sieht so aus:

```
--! Pre: V;
proc(a, b, c);
--! Post: N;
```

Dabei ist *V* die Vorbedingung und *N* die Nachbedingung des Aufrufs, *a* die Liste der aktuellen in-Parameter, *b* die Liste der aktuellen in-out-Parameter und *c* die Liste der aktuellen out-Parameter. Die Typen der einander entsprechenden formalen und aktuellen Parameter müssen kompatibel sein. *a* ist eine Liste von definierten Ausdrücken ohne Seiteneffekt, *b* eine Liste von initialisierten Variablen (eventuell mit Selektoren) und *c* eine Liste von Variablen (eventuell mit Selektoren). Die Wirkung des Prozeduraufrufs ist dann (bei *copy in / copy out*): beim Aufruf werden die Werte von *a* und *b* simultan *x* und *y* zugewiesen, der Prozedurrumpf *S* ausgeführt und die Werte von *y* und *z* simultan *b* und *c* zugewiesen. Die Parameterübergabe erfolgt mit simultaner Zuweisung, d.h. die Semantik muß unabhängig von der Reihenfolge der Zuweisungen sein

$$x, y := a, b;\ S;\ b, c := y, z;\ .$$

Damit gilt

Definition 3.29: $[wp(proc(a,b,c), N) \equiv wp(x, y := a, b; S; b, c := y, z;\ , N)]$.

•

Um die schwächste Vorbedingung des Aufrufs zu berechnen, braucht man also den Prozedurrumpf. Aber beim Beweis der Korrektheit des Aufrufs soll nur die Spezifikation der Prozedur benutzt werden, weil der Prozedurrumpf nicht bekannt sein kann oder um den Beweis der Korrektheit der Prozedur wieder zu verwenden. Dabei geht man davon aus, daß die Prozedur selbst korrekt ist. Analog zur Programmentwicklung, bei der Prozeduren wieder verwendet werden sollen, soll auch beim Programmbeweis ein schon erbrachter Beweis wieder verwendet und nicht noch einmal geführt werden. Mathematisch entspricht dieses Vorgehen der Verwendung eines Lemmas beim Beweis eines Satzes, wobei das Lemma die Spezifikation und der Satz der zu beweisende Aufruf ist. Da man den Prozedurrumpf nicht kennt, ist es im Allgemeinen nicht möglich, die schwächste Vorbedingung *wp* des Aufrufs zu berechnen. (Die schwächste Vorbedingung $wp(S, P)$ ist eine Funktion mit 2 Argumenten. Schon daran sieht man, daß sie im Allgemeinen nicht ohne Kenntnis des 1. Argumentes berechnet werden kann. Statt des Rumpfes *S* hat man hier zwar die Spezifikation der Prozedur, aber daraus ist *S* nicht eindeutig bestimmt, da es im Allgemeinen mehrere verschiedene Rümpfe *S* gibt, die eine Spezifikation erfüllen. Die verschiedenen *wp* können dann sogar unvergleichbar (Definition 2.36) sein, wie das folgende Beispiel zeigt:

```
-- P: y = 0 and x = 0
-- R: y = x
```

Diese Spezifikation wird sowohl von `x := 0` als auch von `y := 0` erfüllt. Die beiden *wp*s sind $y = 0$ und $x = 0$, doch gilt weder $[x = 0 \Rightarrow y = 0]$ noch $[y = 0 \Rightarrow x = 0]$.)

Man muß eine stärkere Vorbedingungen p (d.h. $p \Rightarrow wp$) und damit eine stärkere, also approximierte VC ($V \Rightarrow p$) berechnen. Falls so eine VC gilt, ist der Prozeduraufruf korrekt. Da aber keine schwächste Vorbedingung verwendet wird, gilt die Umkehrung eventuell nicht. D.h. ein Prozeduraufruf kann korrekt sein, ohne daß die VC gilt. So eine VC verwendet eine Vorbedingung, die ohne Rumpf ermittelt wird. Da sie im Allgemeinen nicht die schwächste Vorbedingung ist, also stärker als eigentlich nötig, kann es sein, daß sie so stark ist, daß sie nicht von der Vorbedingung des Aufrufs impliziert wird, obwohl der Aufruf korrekt ist, wie Beispiel 7 weiter unten zeigt. Es ist allerdings möglich, die schwächste Vorbedingung des Aufrufs nur mit Kenntnis der Spezifikation der Prozedur zu berechnen (unter der Voraussetzung der Korrektheit des Prozedurrumpfes) [BMW89]. Von verschiedenen VCs wird hier nur die verwendet, die die schwächst mögliche Vorbedingung nur mit Kenntnis der Spezifikation der Prozedur liefert.

Funktionen werden als spezielle Prozeduren behandelt und ebenso wie globale Variablen, die durch die Prozedur verändert werden, vor Anwendung der VC eliminiert. Diese Elimination wird im folgenden Abschnitt erläutert.

3.4.3.1 Elimination von Funktionen und globalen Variablen

Funktionen kann man als spezielle Prozeduren betrachten, die genau einen out-Parameter und keinen in-out-Parameter sowie beliebig viele (auch 0) in-Parameter haben. Die Rückgabe des Wertes erfolgt durch mindestens 1 return-Statement im Funktionsrumpf. Die Funktionsdefinition sieht so aus:

```
function f(x) return T is
   Def
begin
   StatF
end f;
```

x ist die Liste der formalen in-Parameter, *T* ist der Typ der Funktion, `Def` der Definitionsteil und `StatF` der Anweisungsteil. Er enthält $n \geq 1$ return-Statements **`return`** `e`$_i$, wobei $e_1 \ldots e_n, n \geq 1$ Ausdrücke vom Typ *T* sind. Diese Funktionsdefinition wird in folgende Prozedurdefinition transformiert:

```
procedure fp(x: in; z: out T) is
  Def
begin
  StatFP
end fp;
```

`StatFP` ergibt sich aus `StatF` durch Substitution von **`return`** e_i durch die Zuweisung `z :=` e_i. Neben der Funktionsdefinition muß auch der Funktionsaufruf transformiert werden. Hier besteht ein wesentlicher Unterschied zum Prozeduraufruf. Während der Prozeduraufruf eine Anweisung ist, ist der Funktionsaufruf ein Ausdruck *E* bzw. ein Teil davon. Die Transformation des Funktionsaufrufs `f(a)` besteht im Aufruf `fp(a,h)` unmittelbar vor der Anweisung, die *E* enthält, und der Substitution von `f(a)` in *E* durch *h*, wobei *h* eine neue Variable ist. Diese Transformation wird solange wiederholt, bis kein Funktionsaufruf mehr vorhanden ist.
Die Reihenfolge, in der die Substitutionen durchgeführt werden, falls eine Anweisung mehrere Aufrufe der gleichen Funktion enthält, spielt keine Rolle. Bei der Transformation geschachtelter Aufrufe entstehen zwar Datenabhängigkeiten der neuen Variablen, aber durch den Aufruf von `fp(a, h)` *unmittelbar* vor der Anweisung, die *E* enthält, ist sichergestellt, daß alle Variablen vor einem lesenden Zugriff definiert sind. Daher spielt es auch keine Rolle, ob geschachtelte Aufrufe von innen nach außen oder von außen nach innen oder anders bearbeitet werden.

Beispiel:

```
function f(x, y: integer) return integer is
begin
  if x > y
  then return x – y,
  else return x + 1;
  end if;
end f;
```

wird transformiert in

```
procedure fp(x, y: in integer; z: out integer) is
begin
  if x > y
  then z := x - y;
else z := x + 1;
end if;
 end fp;
```

Die Zuweisung `A(f(c,b-1)+1) := f(v-1,f(v, w)) + f(w - v, v);` wird in 4 Schritten transformiert:

1. Schritt:

```
fp(c,b-1,h1);
A(h1+1) := f(v-1,f(v, w)) + f(w - v, v);
```

2. Schritt:

```
fp(c,b-1,h1);
fp(v-1,f(v, w),h2);
A(h1+1) := h2 + f(w - v, v);
```

3. Schritt:

```
fp(c,b-1,h1);
fp(v,w,h3);
fp(v-1,h3,h2);
A(h1+1) := h2 + f(w - v, v);
```

4. Schritt:

```
fp(c,b-1,h1);
fp(v,w,h3);
fp(v-1,h3,h2);
fp(w-v,v,h4);
```

```
A(h1+1) := h2 + h4;
```

•

Nach der Elimination von Funktionsaufrufen werden globale Variablen *g*, die im Prozedurrumpf *S* geändert werden, durch Einführung neuer Parameter eliminiert. Globale Variablen, die nur gelesen werden, brauchen nicht berücksichtigt zu werden, da sie keine Seiteneffekte verursachen können.
Eine globale Variable *g*, die in *S* nur geschrieben, aber nicht gelesen wird, wird durch einen neuen out-Parameter ersetzt. Die Änderung von *g* wird durch eine *einzige* Zuweisung modelliert. Für alle anderen Anweisungen *S* gilt dann $trans(S, g)$. Das ist möglich, da eine Änderung von *g* keine Datenabhängigkeiten zu anderen Variablen verursacht, weil *g* nach einer Zuweisung nie gelesen wird. Der Rumpf von *P* kann als eine Sequenz von Anweisungen $S_1;\ldots;S_n$, die *g* nicht verändern und solchen, die *g* verändern betrachtet werden, also

```
procedure P(x: in Typ; y: in out Typ; z: out Typ) is
  Def l;
begin
  S1;
g := e1(x, y, l);
   :
g := en-1(x, y, l);
  Sn-1;
g := en(x, y, l);
        :
  Sn;
end P;
```

Alle Zuweisungen zu *g* bis auf die letzte können weggelassen werden, so daß *P* dann so aussieht:

```
procedure P(x: in Typ; y: in out Typ; z: out Typ) is
  Def l;
begin
  S1;
   :
  Sn-1;
  g := en(x, y, l);
   :
  Sn;
end P;
```

Dann gilt

Lemma 3.30: Die beiden Rümpfe

```
S1;
g := e1(x, y, l);
          :
g := en-1(x, y, l);
Sn-1;
g := en(x, y, l);
```

Und

```
S1;
:
Sn-1;
g := en(x, y, l);
```

sind semantisch äquivalent.

Beweis: gezeigt wird die semantische Äquivalenz von

```
g := e1(x, y, l); S; g := e2(x, y, l);
```
und

```
S; g := e2(x, y, l);.
```

Das Lemma folgt dann durch Induktion. Zu zeigen ist nach Satz 2.31

$[wp(g := e1(x,y,l); S; g := e2(x,y,l), N) \equiv wp(S; g := e2(x,y,l), N)]$ (N ist allquantifiziert)

$wp(g := e1(x,y,l); S; g := e2(x,y,l), N)$

≡ Zuweisungsregel

$wp(g := e1(x,y,l), wp(S, N^{g}_{e2(x,y,l)}))$

≡ Zuweisungsregel

$wp(S, N^{g}_{e2(x,y,l)})^{g}_{e1(x,y,l)}$

≡ Implikationskette

$g \notin Var(e2(x,y,l)) \wedge g \notin Var(S) \Rightarrow g \notin Var(N^{g}_{e2(x,y,l)}) \wedge g \notin Var(S) \Rightarrow g \notin Var(wp(S, N^{g}_{e2(x,y,l)}))$

$wp(S, N^{g}_{e2(x,y,l)})$

≡ Zuweisungsregel

$wp(S; g := e2(x,y,l), N)$

•

Die globalen Variablen *g* werden nun in *S* durch neue out-Parameter *wg* ersetzt, so daß man folgende Transformation erhält:

```
procedure P(x: in Typ; y: in out Typ; z: out Typ) is
   Def l;
begin
   S1;
   g := e;
   S2;
end P;
```

wird transformiert in

```
procedure P(x: in Typ; y: in out Typ; z, wg: out Typ) is
   Def l;
begin
   S1;
   wg := e;
   S2;
end P;
```

Dann gilt

Lemma 3.31:

Der Aufruf `P(a, b, c)` ist semantisch äquivalent zu dem Aufruf `P(a, b, c, g)`.

Beweis: zu zeigen ist nach Satz 2.31 $[wp(P(a,b,c), N) \equiv wp(P(a,b,c,g), N)]$ (N ist allquantifiziert)

$wp(P(a,b,c), N)$

≡ Definition 3.29

$wp(a,b := x,y; S_1; g := e; S_2; b,c := y,z; , N)$

≡ Sequenzregel

$wp(a,b := x,y, wp(S_1, wp(g := e, wp(S_2; b,c := y,z; , N))))$

$\equiv$ Zuweisungsregel

$wp(a,b \coloneqq x,y, wp(S_1, wp(S_2, (N^{b,c}_{y,z})^{g}_{e})))$

$\equiv$ Substitutionslemma

$wp(a,b \coloneqq x,y, wp(S_1, wp(S_2, N^{b,c,g}_{y,z,e}))$

und

$wp(P(a,b,c,g), N)$

$\equiv$ Definition 3.29

$wp(a,b \coloneqq x,y; S_1; wg \coloneqq e; S_2; b,c,g \coloneqq y,z,wg;, N)$

$\equiv$ Sequenzregel

$wp(a,b \coloneqq x,y, wp(S_1, wp(wg \coloneqq e, wp(S_2; b,c,g \coloneqq y,z,wg;, N))))$

$\equiv$ Zuweisungsregel

$wp(a,b \coloneqq x,y, wp(S_1, wp(S_2, (N^{b,c,g}_{y,z,wg})^{wg}_{e})))$

$\equiv$ Substitutionslemma

$wp(a,b \coloneqq x,y, wp(S_1, wp(S_2, N^{b,c,g}_{y,z,e}))$

•

Eine globale Variable *g*, die in *S* geschrieben und gelesen wird, wird durch einen neuen in-out-Parameter *xg* ersetzt.

```
procedure P(x: in Typ; y: in out Typ; z: out Typ) is
   Def
begin
   S
end P;
```

wird transformiert in

```
procedure P(x: in Typ; y, xg: in out Typ; z: out Typ) is
   Def
begin
   S^g_xg
end P;
```

Dann gilt

Lemma 3.32:

Der Aufruf `P(a, b, c)` ist für $b \neq g$ semantisch äquivalent zu dem Aufruf `P(a, b, g, c)`.

Beweis: zu zeigen ist nach Satz 2.31 $[wp(P(a,b,c),N) \equiv wp(P(a,b,g,c),N)]$ (N ist allquantifiziert)

$wp(P(a,b,c), N)$

$\equiv$ Definition 3.29

$wp(x,y \coloneqq a,b; S; b,c \coloneqq y,z;\ N)$

$\equiv$ Sequenz, Zuweisungsregel

$wp(S, N^{b,c}_{y,z})^{x,y}_{a,b}$

und

$$wp(P(a,b,g,c),N)$$

≡ Definition 3.29

$$wp(x,y,xg := a,b,g;S^{g}_{xg};b,g,c := y,z,xg;,N)$$

≡ Sequenz, Zuweisungsregel

$$wp(x,y,xg := a,b,g;,wp(S^{g}_{xg},N^{b,c,g}_{y,z,xg}))$$

≡ Substitutionslemma für *wp*, $xg \notin Var(S) \cup free(N)$

$$wp(x,y,xg := a,b,g;,wp(S,N^{b,c}_{y,z})^{g}_{xg})$$

≡ Zuweisungsregel

$$(wp(S,N^{b,c}_{y,z})^{g}_{xg})^{x,y,xg}_{a,b,g}$$

≡ Substitutionslemma ($b \neq g$)

$$wp(S,N^{b,c}_{y,z})^{x,y}_{a,b}$$

•

Nach diesen beiden Transformationen hat man nur noch Prozeduren ohne Seiteneffekte. Damit sind die Voraussetzungen für die Konstruktion der Vorbedingung nach [BMW89] erfüllt.

3.4.3.2 Die schwächst mögliche Vorbedingung eines Prozeduraufrufs

Die schwächst mögliche Vorbedingung *vcb* eines Prozeduraufrufs ohne Kenntnis des Rumpfes ist ein Prädikat, das von der Nachbedingung *N* des Aufrufs und statt des Rumpfes *S* selbst von dessen Spezifikation (*P*, *R*) abhängt.

Satz 3.33 [BMW89]: Das folgende Prädikat *vcb*(*P*, *R*, *N*) ist die schwächste Vorbedingung für den Prozeduraufruf, wobei man nur die Spezifikation, aber nicht den Rumpf der Prozedur kennt.

$$(\exists s : P^{x,y}_{a,b}) \wedge (\forall y,z : (\forall s : P^{x,y}_{a,b} \Rightarrow R^{x}_{a}) \Rightarrow N^{b,c}_{y,z}).$$

Dabei sind *s* die Spezifikationsvariablen.
Beweis: Zum Beweis sind 2 Teile zu zeigen:
vcb ist Vorbedingung und falls ein anderes Prädikat eine Vorbedingung ist, dann ist es nicht schwächer. Der Beweis ist in [BMW89]

•

Bemerkung: In [B93] wird ein einfacheres Prädikat als *vcb* angegeben, das unter einer bestimmten Voraussetzung auch die schwächste Vorbedingung für den Prozeduraufruf ohne Kenntnis des Rumpfes liefert, also äquivalent zu *vcb* ist. Als Voraussetzung muß die Spezifikation *normal* im Sinne der Definition in [B93] sein, d.h. $[(Ns :: P) \leq 1]$. Nach [B93] ist diese Voraussetzung im Allgemeinen erfüllt. Man könne zwar Beispiele konstruieren, die diese Voraussetzung nicht erfüllen, diese seien dann jedoch sehr akademisch. Allerdings gibt es realistische und praktische Beispiele, in denen die Vorbedingung der Prozedur *P* die Spezifikationsvariablen *s* in einer Relation enthält, die von mehreren Belegungen von *s* erfüllt werden kann, so daß $[(Ns :: P) \leq 1]$ nicht gilt. Das ist vermutlich in komplexeren Beispielen keine Ausnahme. Außerdem gibt es zu jeder Spezifikation eine äquivalente normale Spezifikation. Diese ist jedoch komplizierter als die nicht normalisierte. Das Problem dieser Vorbedingung besteht zudem darin, daß die Normalität der Spezifikation im Allgemeinen nicht entscheidbar ist, allerdings oft durch eine gewisse menschliche Einsicht leicht gesehen werden kann. Da

das Ziel dieser Arbeit unter anderem eine möglichst weitgehende Automatisierung der Verifikation ist, wird im weiteren Verlauf das Prädikat *vcb* verwendet.

•

vcb wird zunächst mit anderen Vorbedingungen für Prozeduraufrufe aus der Literatur verglichen. Betrachtet werden die Vorbedingungen aus [G81: 153] (*vcg*), aus [M83] (*vcm*), aus [F89: 137] (*vcf*) und *vcb*.

$vcg \equiv P_{a,b}^{x,y} \wedge (\forall u,v : R_{a,u,v}^{x,y,z} \Rightarrow N_{u,v}^{b,c})$

$vcm \equiv (A \wedge P)_{a,b}^{x,y}$, wobei die Adaption *A* ein möglichst schwaches Prädikat ist, das $A \wedge R \Rightarrow N_{y,z}^{b,c}$ erfüllt

$vcf \equiv (\exists s : P_{a,b}^{x,y} \wedge (\forall u,v : R_{a,u,v}^{x,y,z} \Rightarrow N_{u,v}^{b,c}))$

Satz 3.34: Die 3 Prädikate *vcg*, *vcf* und *vcb* bilden eine Kette. Es gilt

$vcg \equiv vcm \Rightarrow vcf \Rightarrow vcb$ und $vcg \neq vcf \wedge vcf \neq vcb$.

Beweis:

1. [$vcg \equiv vcm$]: Beweis in [BMW86]; ein Problem mit *vcm* ist, daß die Adaption angegeben werden muß und im Allgemeinen nicht automatisch ermittelt werden kann.

2. [$vcg \Rightarrow vcf$]:

$$[P_{a,b}^{x,y} \wedge (\forall u,v : R_{a,u,v}^{x,y,z} \Rightarrow N_{u,v}^{b,c}) \Rightarrow (\exists s : P_{a,b}^{x,y} \wedge (\forall u,v : R_{a,u,v}^{x,y,z} \Rightarrow N_{u,v}^{b,c}))]$$

vcg ist ein Prädikat mit den freien Variablen *s*, *a* und *b*. *vcf* ist ein Prädikat mit den freien Variablen *a* und *b*, so daß man schreiben kann

$[(\forall a,b,s : P(s)_{a,b}^{x,y} \wedge (\forall u,v : R(s)_{a,u,v}^{x,y,z} \Rightarrow N_{u,v}^{b,c}) \Rightarrow (\exists s' : P(s')_{a,b}^{x,y} \wedge (\forall u,v : R(s')_{a,u,v}^{x,y,z} \Rightarrow N_{u,v}^{b,c})))]$

$\equiv$ mit $s' = s$

$[(\forall a,b,s : P(s)_{a,b}^{x,y} \wedge (\forall u,v : R(s)_{a,u,v}^{x,y,z} \Rightarrow N_{u,v}^{b,c}) \Rightarrow P(s)_{a,b}^{x,y} \wedge (\forall u,v : R(s)_{a,u,v}^{x,y,z} \Rightarrow N_{u,v}^{b,c}))]$

$\equiv$

true

Der Unterschied zwischen *vcg* und *vcf* liegt nur darin, daß in *vcg* die Spezifikationsvariablen *s* nicht explizit quantifiziert, d.h. allquantifiziert sind und in *vcf* existenzquantifiziert. Daß dieser Unterschied dafür sorgt, daß *vcg* und *vcf* nicht äquivalent sind, wird durch folgendes Beispiel gezeigt.

3. $vcg \neq vcf$: Beweis durch Gegenbeispiel eines korrekten Prozeduraufrufs, der mit *vcf*, aber nicht mit *vcg* bewiesen werden kann:

```
--! P: x = k1 ∧ y = k2;
--! R: x = k2 ∧ y = k1;
procedure swap(x, y: in out integer);
```

Folgender Prozeduraufruf wird mit *vcf*, jedoch nicht mit *vcg* bewiesen:

```
--! V: a<=b;
swap(a, b);
--! N: a>=b;
```

Die Vorbedingung für Prozeduraufrufe aus [G81: 153] lautet $P_{a,b}^{x,y} \wedge (\forall u,v : R_{u,v}^{y,z} \Rightarrow N_{u,v}^{b,c})$. Auf das Beispiel angewendet ergibt das die VC

$[V \Rightarrow vcg]$

$\equiv$

$[a \le b \Rightarrow a = k1 \wedge b = k2 \wedge (\forall u, v : u = k2 \wedge v = k1 \Rightarrow u \ge v)]$

$\equiv$

false

aber

$[V \Rightarrow vcf\,]$

$\equiv$

$[a \le b \Rightarrow (\exists k1, k2 : a = k1 \wedge b = k2 \wedge (\forall u, v : u = k2 \wedge v = k1 \Rightarrow u \ge v))]$

$\equiv$ one-point rule

$[a \le b \Rightarrow (\exists k1, k2 : a = k1 \wedge b = k2 \wedge k2 \ge k1)]$

$\equiv$ one-point rule

$[a \le b \Rightarrow b \ge a]$

$\equiv$

true

4. [*vcf* $\Rightarrow$ *vcb*]: um die Lesbarkeit des Beweises zu erhöhen, wird die Abhängigkeit der Prädikate *P* und *R* von *s* explizit angegeben.

$[(\exists s : P(s)^{x,y}_{a,b} \wedge (\forall u, v : R(s)^{x,y,z}_{a,u,v} \Rightarrow N^{b,c}_{u,v})) \Rightarrow (\exists s : P(s)^{x,y}_{a,b}) \wedge (\forall y, z : (\forall s : P(s)^{x,y}_{a,b} : R(s)^{x}_{a}) : N^{b,c}_{y,z})]$
$\equiv$ Skolemisierung, gebundene Umbenennung

$[(\forall s : P(s)^{x,y}_{a,b} \wedge (\forall u, v : R(s)^{x,y,z}_{a,u,v} \Rightarrow N^{b,c}_{u,v}) \Rightarrow (\exists s' : P(s')^{x,y}_{a,b}) \wedge (\forall y, z : (\forall s' : P(s')^{x,y}_{a,b} : R(s')^{x}_{a}) : N^{b,c}_{y,z}))]$
$\equiv$ mit $s' = s$

$[(\forall s : P(s)^{x,y}_{a,b} \wedge (\forall u, v : R(s)^{x,y,z}_{a,u,v} \Rightarrow N^{b,c}_{u,v}) \Rightarrow P(s)^{x,y}_{a,b} \wedge (\forall y, z : (\forall s' : P(s')^{x,y}_{a,b} : R(s')^{x}_{a}) : N^{b,c}_{y,z}))]$

$\equiv$ Skolemisierung, portation

$[(\forall x, y, s : P(s)^{x,y}_{a,b} \wedge (\forall u, v : R(s)^{x,y,z}_{a,u,v} \Rightarrow N^{b,c}_{u,v})) \wedge (\forall s' : P(s')^{x,y}_{a,b} : R(s')^{x}_{a}) \Rightarrow N^{b,c}_{y,z})]$

$\Leftarrow$ mit $s' = s,\ u = y,\ v = z$

$[(\forall x, y, s : P(s)^{x,y}_{a,b} \wedge (R(s)^{x}_{a} \Rightarrow N^{b,c}_{y,z}) \wedge (P(s)^{x,y}_{a,b} \Rightarrow R(s)^{x}_{a}) \Rightarrow N^{b,c}_{y,z})]$

$\Leftarrow$ 2 mal modus ponens

true

5. *vcf* $\neq$ *vcb* : Beweis durch Gegenbeispiel eines korrekten Prozeduraufrufs, der mit *vcb*, aber nicht mit *vcf* bewiesen werden kann (*s* ist eine Spezifikationsvariable):

```
-- P: x≤s≤y
procedure p(x: in; y: in out; z: out)
-- R: y>s ∨ s>z
```

Diese Prozedurspezifikation wird z.B. von `z := x-1; y := y+1` erfüllt. Folgender Prozeduraufruf wird mit *vcb*, jedoch nicht mit *vcf* bewiesen:

```
-- b = 100
p(0,b,c);
-- 0>b ∨ b>c ∨ c>100
```

Mit *vcb* ergibt sich die VC

$[b = 100 \Rightarrow (\exists s : 0 \le s \le b) \wedge (\forall y, z : (\forall s : 0 \le s \le b : y > s \vee s > z) : 0 > y \vee y > z \vee z > 100)]$

≡ Substitution, Logik

$[(\exists s : 0 \le s \le 100) \wedge (\forall y, z : (\exists s : 0 \le s \le 100 : y \le s \le z) \vee 0 > y \vee y > z \vee z > 100)]$

≡ mit $s = 0$, Logik

$[0 \le 0 \le 100 \wedge (\forall y, z : 0 \le y \le z \le 100 : (\exists s : 0 \le s \le 100 : y \le s \le z))]$

≡ mit $s = y$

$[(\forall y, z : 0 \le y \le z \le 100 : 0 \le y \le 100 \wedge y \le y \le z)]$

≡ Logik

true

aber mit *vcf* ergibt sich die VC

$[b = 100 \Rightarrow (\exists s : 0 \le s \le b \wedge (\forall u, v : u > s \vee s > v : 0 > u \vee u > v \vee v > 100))]$

≡ Substitution

$[(\exists s : 0 \le s \le 100 \wedge (\forall u, v : u > s \vee s > v \Rightarrow 0 > u \vee u > v \vee v > 100))]$

≡ Kontraposition

$[(\exists s : 0 \le s \le 100 \wedge (\forall u, v : 0 \le u \le v \le 100 : u \le s \le v))]$

≡ Logik

false

•

Die mit *vcb* gebildete VC $[V \Rightarrow vcb]$ wird anhand einiger Beispiele erläutert, von denen einige aus [G81: 161-162] sind:

Beispiele:

Beispiel 1: Vertauschen von zwei Integervariablen:

Spezifikation:

```
--! P: x = k1 ∧ y = k2;
--! R: x = k2 ∧ y = k1;
procedure swap(x, y: in out integer);
```

Aufruf:

```
--! V: a = 3 * n ∧ b = k - 2;
swap(a, b);
--! N: a<k;
```

Beweis:

$[V \Rightarrow vcb]$

≡

$[a = 3n \wedge b = k - 2 \Rightarrow (\exists k1, k2 : a = k1 \wedge b = k2) \wedge$
$(\forall x, y : (\forall k1, k2 : a = k1 \wedge b = k2 \Rightarrow x = k2 \wedge y = k1) \Rightarrow x < k)]$

≡

true

•

Beispiel 2: Potenzierung:

Spezifikation:

```
--! P: y = Y;
--! R: y = x ** Y;
procedure pow(x: in integer; y in out integer);
```

Aufruf:

```
--! V: i >= 1 ∧ t >= 3;
pow(i+1, t);
--! N: i >= 1 ∧ t >= 8;
```

Beweis:

$[V \Rightarrow vcb]$

$\equiv$

$[i \geq 1 \wedge t \geq 3 \Rightarrow (\exists Y : t = Y) \wedge$
$(\forall y : (\forall Y : t = Y \Rightarrow y = (i+1) ** Y) \Rightarrow i \geq 1 \wedge y \geq 8)]$

$\equiv$

true

•

Beispiel 3:

Spezifikation:

```
--! P: true;
--! R: z = 1 ∨ z = 2;
procedure p(z: out integer);
```

Aufruf:

```
--! V: c ∈ {0,...,10};
p(c);
--! N: c = 2;
```

Beweis:

$[V \Rightarrow vcb]$

$\equiv$

$[c \in \{0,\ldots,10\} \Rightarrow (\forall z : z = 1 \vee z = 2 \Rightarrow z = 2)]$

$\equiv$

false

•

Beispiel 4: Absteigend ordnen:

Spezifikation:

```
--! P: a = k1 ∧ b = k2;
--! R: a = max(k1, k2) ∧ b = min(k1, k2);
```

```
procedure desc(a, b: in out integer);
```

Aufruf:

```
--! V: true;
desc(x, y);
--! N: x >= y;
```

Beweis:

$[V \Rightarrow vcb]$

$\equiv$

$[(\exists k1, k2 : x = k1 \wedge y = k2) \wedge$

$(\forall a, b : (\forall k1, k2 : x = k1 \wedge y = k2 \Rightarrow a = \max(k1, k2) \wedge b = \min(k1, k2)) \Rightarrow a \geq b)]$

$\equiv$

true

•

Beispiel 5:

Spezifikation:

```
--! P: y = Y;
--! R: y = x ∧ u = x - Y ∧ v = x + Y;
procedure set(x: in integer; y: in out integer; u, v: out integer);
```

Aufruf:

```
--! Pre: b - 2 >= 3 * a;
set(3 * a + 2, b, c, d);
--! Post: d - c >= 6 * a + 4;
```

Beweis:

$[V \Rightarrow vcb]$

$\equiv$

$[b - 2 \geq 3a \Rightarrow (\exists Y : b = Y) \wedge$

$(\forall y, u, v : (\forall Y : b = Y \Rightarrow y = 3a + 2 \wedge u = 3a + 2 - Y \wedge v = 3a + 2 + Y) \Rightarrow v - u \geq 6a + 4)]$

$\equiv$

true

•

Beispiel 6: Ganzzahlige Wurzel:

Spezifikation:

```
--! P: x >= 0;
--! R: z >= 0 ∧ z ** 2 <= x < (z+1)** 2;
procedure sqrt(x: in; z: out);
```

Aufruf:

```
--! V: 1 < a <= 2p;
sqrt(a, b);
--! N: 1 <= b <= p;
```

Beweis:

$[V \Rightarrow vcb]$

$\equiv$

$[1 \le a \le 2p \Rightarrow a \ge 0 \wedge (\forall z : (a \ge 0 \Rightarrow z^2 \le a < (z+1)^2 \wedge z \ge 0) \Rightarrow 1 \le z \le p)]$

$\equiv$

true

•

Beispiel 7: Korrekter, aber nicht verifizierbarer Aufruf:

Spezifikation:

```
--! P: m-1 ≤ y ≤ m;
--! R: z ≤ m ≤ y;
procedure p(y: in out; z: out) is
begin
y := y+1;
z := y-1;
end p;
```

Die Korrektheit des Prozedurrumpfes ergibt sich aus der VC

$[m-1 \le y \le m \Rightarrow y \le m \le y+1]$

$\equiv$

$[y = m \vee y = m-1 \Rightarrow y \le m \le y+1]$

$\equiv$

true

Aufruf:

```
--! V: b < 0;
p(b, c);
--! N: b + c ≤ 0;
```

Mit Definition 3.29 ergibt sich als VC für den Aufruf (mit Kenntnis des Rumpfes)

$[b < 0 \Rightarrow 2b < 0] \equiv true$

Aber

$[V \Rightarrow vcb]$

$\equiv$

$[b < 0 \Rightarrow (\exists m : m-1 \le b \le m) \wedge (\forall y, z : (\forall m : m-1 \le b \le m \Rightarrow z \le m \le y) \Rightarrow z + y \le 0)]$

$\equiv$

false

Beweis durch das Gegenbeispiel $b = -1, z = -1, y = 2$, womit sich ergibt

$true \wedge ((\forall m : m-1 \le -1 \le m \Rightarrow -1 \le m \le 2) \Rightarrow false)$

$\equiv$ Logik

$(\exists m : m-1 \le -1 \le m \wedge (-1 > m \vee m > 2))$

$\equiv$

false

•

Mit *vcb* ist es möglich, alle korrekten Beispielaufrufe außer Beispiel 7 zu beweisen. Beispiel 7 zeigt explizit, daß auch *vcb* nicht die schwächste Vorbedingung für einen Aufruf liefert. Der nicht korrekte Aufruf aus Beispiel 3 kann nicht bewiesen werden, was auch so sein soll.
An dieser Stelle wird noch einmal zu den Schleifen zurückgekehrt. Es geht um eine leistungsfähige Verifikationsregel für Schleifen im Kontext einer umgebenden Anweisung , wofür die Beweisregel für Prozeduren benutzt werden kann.

3.4.4 Eine Vorbedingung für Schleifen im Kontext einer umgebenden Anweisung

Für die Verifikation isolierter Schleifen gibt es die Regeln aus des Abschnitten 3.4.1 und 3.4.2. Falls eine Schleife z.B. innerhalb einer If-Anweisung vorkommt, sollte das keine weiteren Schwierigkeiten machen. Es gibt ja Regeln für die Behandlung zusammengesetzter Anweisungen. Daher wird dieser Fall in der Literatur nicht gesondert behandelt. Aber in diesen Fällen (geschachtelte Schleifen, Schleifen in If- oder Case-Anweisung) kann es wegen der Beweisregeln für die äußeren Anweisungen notwendig sein, die schwächste Vorbedingung der Schleife zu berechnen. Im Falle einer While-Schleife in einer If-Anweisung

```
-- V;
if B then
   while C loop S; end loop;
end if;
-- N;
```

ergibt sich

$$[V \Rightarrow B \wedge wp(while, N) \vee \neg B \wedge N].$$

Da die Berechnung von $wp(while, N)$ im Allgemeinen nicht möglich ist, muß man statt dessen eine andere Vorbedingung verwenden. Man könnte z.B. einfach die in der Spezifikation der Schleife gegebene Vorbedingung `pre` verwenden unter der Voraussetzung, daß `post` die Nachbedingung *N* impliziert, wie im folgenden Beispiel:

```
--!pre1: y>0 and x>=0;
if x<y
then
 q := 1;
else
 q := 0;
 r := x;
 --!pre2: y>0 and x>=0 and x>=y and q=0 and r = x;
 --!post2: 0<=r and r<y<=x and q * y + r = x;
 --!inv: 0<=r and 0<y<=x and q * y + r = x;
 --!term: r;
 while r>=y loop
       r := r - y;
       q := q+1;
 end loop;
end if;
--!post1: (x>=y and 0<=r and r<y and q * y + r = x) or (x<y and
q=1);
```

Die isolierte While-Schleife ist korrekt (der Korrektheitsbeweis wird nicht aufgeführt), und es gilt $[post2 \Rightarrow post1]$. Für die Korrektheit der Schleife im Kontext der If-Anweisung ergibt sich die VC

$$[pre1 \Rightarrow B \wedge wp(Then - Zweig, post1) \vee \neg B \wedge wp(Else - Zweig, post1)]$$

≡

$$[y > 0 \wedge x \geq 0 \Rightarrow x < y \wedge (x \geq y > r \geq 0 \wedge r < y \wedge y + r = x \vee x < y)$$
$$\vee\, x \geq y \wedge wp(q := 0; r := x, wp(while, x \geq y > r \geq 0 \wedge qy + r = x \vee x < y \wedge q = 1))]$$

Da die While-Schleife korrekt ist und $[post2 \Rightarrow post1]$ gilt, kann man für $wp(while, N)$ einfach `pre2` nehmen, womit sich ergibt:

$$[x \geq y > 0 \wedge x \geq 0 \Rightarrow x \geq y > 0 \wedge x \geq 0] \equiv true.$$

In diesem Fall funktioniert diese einfache Methode. Aber wie man sieht, daß `post2` N implizieren. Aber es gibt korrekte Programme, bei denen das nicht gilt, wie Beispiel 2 weiter unten zeigt (Berechnung von $p = a \cdot b$). Aber auch allein die Voraussetzung, daß `post2` N impliziert, ist nicht hinreichend, um den Korrektheitsbeweis führen zu können. Das führt dann dazu, daß ein korrektes Programm nicht als korrekt bewiesen werden kann, wie im folgenden Beispiel:

```
--!pre1: i = n and j = m;
if i >= 0 then
   --!pre2: i = a = abs(n) and j = b = m and i >= 0;
   --!post2: i = 0 and j = a + b and a = abs(n) and b = m;
   --!inv: i >= 0 and i + j = a + b and a = abs(n) and b = m;
   --!term: i;
   while i>0 loop
      j := j+1;
      i := i-1;
   end loop;
else
   j := j - i;
end if;
--!post1: j = m + abs(n);
```

Es gilt zwar $[post2 \Rightarrow post1]$, aber als VC ergibt sich

$$[i = n \wedge j = m \Rightarrow i \geq 0 \wedge i = a = |n| \wedge j = b = m \wedge i \geq 0 \vee i < 0 \wedge j - i = m + |n|]$$

Das kann nicht bewiesen werden, da in `pre2` die Variablen *a* und *b* vorkommen.
Ganz ähnliche Probleme treten bei Prozeduraufrufen auf. Daher liegt es nahe, die Ausführung einer Schleife wie einen Prozeduraufruf zu behandeln. Dabei wird nur die Spezifikation (*pre, post*) der Schleife betrachtet und nicht der Schleifenrumpf. Die Vorbedingung der Schleife ist dann ein Prädikat, das von der Nachbedingung *N* und statt vom Schleifenrumpf selbst von der Spezifikation (*pre, post*) der Schleife abhängt. Man kann deshalb eine ähnliche Beweisregel verwenden wie für Prozeduraufrufe. Es bestehen allerdings zwei Unterschiede:

1. bei Prozeduren gibt es 3 Parameterarten, nämlich **`in`**, **`in out`** und **`out`**. Das entspricht in der Prozedur Variablen, die nicht verändert werden, Variablen, die verändert werden und Variablen, die bei Beginn des Schleifenrumpfes undefiniert sind und am Ende des Schleifenrumpfes definiert. In der Schleife gibt es die letzte Art von Variablen nicht. In Ada kann man zwar innerhalb einer Schleife in einem Deklarationsteil neue Variablen definieren; diese sind aber nach Ausführung der Schleife nicht mehr sichtbar.
2. bei Prozeduren und Aufrufen wird zwischen formalen und aktuellen Parametern unterschieden. Bei der Ausführung einer Schleife, die ähnlich wie ein Prozeduraufruf behandelt werden soll, gibt es diese Unterscheidung nicht.

Um eine Beweisregel herzuleiten, geht man von der für Prozeduraufrufe aus. Der Schleifenrumpf wird zum Prozedurrumpf, indem die Variablen in der Schleife und ihrer Spezifikation umbenannt werden. Dann ist die Beweisregel für Prozeduraufrufe anwendbar. Die Schleifen-

variablen, die nicht verändert werden, heißen *a* und die, die verändert werden, heißen *b*. Sie werden in *x* bzw. *y* umbenannt. Die Spezifikation lautet dann $(pre_{x,y}^{a,b}, post_{x,y}^{a,b})$. Das in die Vorbedingung *vcb* für Prozeduraufrufe eingesetzt ergibt die Vorbedingung *oc* für die Korrektheit der Schleife im Kontext einer umgebenden Anweisung:

$$(\exists m\colon (pre_{x,y}^{a,b})_{a,b}^{x,y}) \wedge (\forall y\colon (\forall m\colon (pre_{x,y}^{a,b})_{a,b}^{x,y} \Rightarrow (post_{x,y}^{a,b})_{a}^{x}) \Rightarrow N_{y}^{b})$$

$\equiv$ Substitutionslemma

$$(\exists m\colon pre) \wedge (\forall y\colon (\forall m\colon pre \Rightarrow post_{y}^{b}) \Rightarrow N_{y}^{b})$$

Auf diese Vorbedingung kann man auch durch das eher intuitive und verbale Argument aus [K96: 16] kommen. In der Beweisregel wird der Unterschied der Variablen *b* vor und nach Ausführung der Schleife widergespiegelt. Dazu werden in der Beweisregel eindeutige Hilfsvariablen verwendet, die sonst nicht vorkommen und nur an dieser Stelle gelten. Dieses Problem wird auch in [F89: 138-139] betrachtet. Das Ergebnis ist die Konsequenzregel II. Wenn man in dieser Regel das schwächere Prädikat *vcb* statt *vcf* verwendet, erhält man *oc*.
Im Allgemeinen hat man im Laufe eines Beweises einer umgebenden Anweisung (Schleife oder Fallunterscheidung) die folgende Situation gegeben:

```
-- Vi
Si
-- Ni
-- Z
```

D.h. die Zusicherung *Z* wurde mit *wp*-Regeln rückwärts berechnet und jetzt ist eine möglichst schwache Vorbedingung der inneren Schleife `Si` gesucht. Vorausgesetzt ist die Korrektheit der inneren Schleife, also $[Vi \Rightarrow wp(Si, Ni)]$. Falls die Anschlußbedingung $[Ni \Rightarrow Z]$ erfüllt ist, gilt wegen der Monotonie von *wp* $[wp(Si, Ni) \Rightarrow wp(Si, Z)]$ und mit der Transitivität ergibt sich $[Vi \Rightarrow wp(Si, Z)]$, so daß man mit *Vi* als einer möglichst schwachen Vorbedingung weiterrechnen kann. Mit *oc* ist man nicht auf die Anschlußbedingung angewiesen. Diese Methode zum Beweis von Schleifen im Kontext einer umgebenden Anweisung hat folgende Vorteile:

1. Eine innere Schleife kann isoliert bewiesen werden. Im 2. Beispiel weiter unten dient die innere Schleife dazu, *p* um *a* zu erhöhen. Das kann einfach wie bei einer Prozedur spezifiziert werden, ohne sich um die Anschlußbedingungen zu kümmern. Dadurch ist es leichter, Anschlußbedingungen zu finden.
2. Es können Schleifen bewiesen werden, die sonst nicht bewiesen werden können, weil die Anschlußbedingungen nicht erfüllt sind. Zwar könnte man die Anschlußbedingungen so gestalten, daß sie erfüllt sind, aber sie sind dann schwieriger zu finden, und die isolierte Betrachtung der inneren Schleife (Vorteil 1) ist dann auch nicht mehr gegeben.

Mit dieser Beweisregel ergibt sich für das obige Beispiel die VC

$$[i = n \wedge j = m \Rightarrow i \geq 0 \wedge$$
$$(\exists a, b : i = a = |n| \wedge j = b = m \wedge i \geq 0) \wedge$$
$$(\forall x, y : (\forall a, b : i = a = |n| \wedge j = b = m \wedge i \geq 0 \Rightarrow x = 0 \wedge y = a + b \wedge a = |n| \wedge b = m) \Rightarrow y = m + |n|)$$
$$\vee i < 0 \wedge j - i = m + |n|]$$

$\equiv$

true

•

Das 2. Beispiel zeigt die Verwendung dieser Beweisregel an einer geschachtelten For-Schleife. Es wird das Produkt *p* der nicht negativen ganzen Zahlen *a* und *b* berechnet. Außerdem soll $T_p(a \cdot b)$ gelten, so daß die Typbedingung immer erfüllt ist und im folgenden Beweis weggelassen werden kann.

```
-- Va: a, b ≥ 0 ∧ p = 0
for i in 1 .. b loop
   -- Vi: p = c ∧ a ≥ 0
   for j in 1 .. a loop
         p := p+1;
         -- Pi: p = c + j
   end loop;
   -- Ni: p = c + a
   -- Pa: p = a * i ∧ a ≥ 0
end loop;
-- Na: p = a * b
```

1. Beweis der Korrektheit der inneren Schleife:

(1) $[1 \le a \wedge p = c \Rightarrow p+1 = c+1] \equiv true$

(2) $[2 \le j \le a \wedge p = c + j - 1 \Rightarrow p+1 = c + j] \equiv true$

(3) $[1 \le a \wedge p = c + a \Rightarrow p = c + a] \equiv true$

(4) $[1 > a \wedge p = c \wedge a \ge 0 \Rightarrow p = c + a] \equiv true$

2. Beweis der Korrektheit der äußeren Schleife:

(1) $[1 \le b \wedge 0 \le a \wedge p = 0 \Rightarrow wp(innere\ Schleife, p = a \cdot 1 \wedge a \ge 0)]$

Die Nachbedingung der inneren Schleife $p = c + a$ impliziert nicht die Invariante der äußeren Schleife $p = a \cdot 1 \wedge a \ge 0$, so daß man jetzt nicht mit der Vorbedingung der inneren Schleife $p = c \wedge a \ge 0$ weiterrechnen kann. Mit *oc* ergibt sich jedoch *true*

(2) $[2 \le i \le b \wedge p = a \cdot (i-1) \wedge a \ge 0 \Rightarrow wp(innere\ Schleife, p = a \cdot i \wedge a \ge 0)] \equiv \text{true}$

(3) $[1 \le b \wedge p = a \cdot b \wedge a \ge 0 \Rightarrow p = a \cdot b] \equiv true$

(4) $[1 > b \wedge 0 \le a, b \wedge p = 0 \Rightarrow p = a \cdot b] \equiv true$

•

Diese Methode eignet sich für alle Anweisungen, für die keine *wp* berechnet werden kann. Man benutzt statt der Anweisung selbst ihre Spezifikation und betrachtet ihre Ausführung wie einen Prozeduraufruf. In diesem Abschnitt wurde das für Schleifen gemacht. Diese Methode zur Verifikation geschachtelter Schleifen ist in FPP implementiert.

3.4.5 Rekursive Prozeduren

Da rekursive Prozeduren eine induktive Implementierung induktiv definierter Prädikate darstellen, liegt es nahe, solche Prozeduren durch Induktion zu beweisen. Der Induktionsanfang wird dabei durch die Zweige (im Allgemeinen mehrere) der Prozedur gegeben, die keinen rekursiven Aufruf enthalten. Die Induktionsannahme ist dann, daß die Prozedur für die aktuellen Parameter in den rekursiven Aufrufen korrekt ist. Unter dieser Annahme wird gezeigt, daß der Prozedurrumpf korrekt ist. Beispiele für dieses Vorgehen findet man z.B. in [H92: 37-44] oder [M83]. Diese Beispiele verwenden zwar das oben erwähnte allgemeine Schema, aber kein rein mechanisches / algorithmisches Vorgehen, d.h. es ist noch einiges an menschlicher

Intuition, Einfällen und „Tricks“ nötig. Doch zur vollständigen Automatisierung des Korrektheitsbeweises rekursiver Prozeduren ist ein rein mechanisches Vorgehen notwendig.
In [HM96] wird ein Verfahren zur automatischen Erzeugung von VCs für indirekt rekursive Prozeduren angegeben. Allerdings kann damit auch nur die partielle Korrektheit gezeigt werden. Da die zugrunde liegende VC für Prozeduren nicht die schwächst mögliche ist, können die so erzeugten VCs für gegenseitig rekursive Prozeduren auch nicht die schwächst möglichen sein. Zudem wird in [HM96] auch kein Beispiel für den Korrektheitsbeweis indirekt rekursiver Prozeduren angegeben.
Nach dem Induktionsprinzip, aber rein mechanisch, ist *vcb* auch auf rekursive Prozeduren anwendbar.

$$vcb \equiv (\exists m : P_{a,b}^{x,y}) \wedge (\forall y, z : (\exists m : P_{a,b}^{x,y} \Rightarrow R_a^x) \Rightarrow N_{y,z}^{b,c})$$

Die prinzipielle Vorgehensweise wird zunächst an einem sehr einfachen Beispiel gezeigt, nämlich der Prozedur `fac(n, f)`, die die Fakultät *f* einer natürlichen Zahl *n* berechnen soll. Um das Wesentliche zu zeigen, wird die Typprüfung weggelassen.

```
procedure fac(n: in natural; f: out positive) is
-- V: n >= 0
-- N: f = n!
begin
   if n = 0 then f := 1;
   else  fac(n-1, f);
         f := f * n;
   end if;
end fac;
```

Als VC ergibt sich

$[n \geq 0 \Rightarrow wp(IF, f = n!)]$

$\equiv$ If-Regel

$[n \geq 0 \Rightarrow n = 0 \wedge 1 = n! \vee n \neq 0 \wedge wp(fac(n-1, f); f := f \cdot n, f = n!)]$

$\equiv$ case analysis

$[(n \geq 0 \wedge n = 0 \Rightarrow 1 = n!) \wedge$
$(n \geq 0 \wedge n \neq 0 \Rightarrow wp(fac(n-1, f); f := f \cdot n, f = n!))]$

$\equiv$ Logik, Zuweisungsregel, Arithmetik

$[n \geq 0 \wedge n \neq 0 \Rightarrow wp(fac(n-1, f), f = (n-1)!)]$

$\Leftarrow$ Logik, Induktionsvoraussetzung, *vcb*

$[n \geq 1 \Rightarrow n-1 \geq 0 \wedge (\forall f : (n-1 \geq 0 \Rightarrow f = (n-1)!) \Rightarrow f = (n-1)!)]$

$\equiv$ Logik

true

•

Allerdings ist dies nur eine Beweisregel für die partielle Korrektheit, da man damit auch die Korrektheit einer theoretisch nicht terminierenden rekursiven Prozedur beweisen kann, wie das folgende, leicht abgeänderte Beispiel zeigt.

```
procedure fac(n: in natural; f: out positive) is
-- V: n >= 0
-- N: f = n!
begin
   if n = 0 then f := 1;
   else  fac(n+1, f);
         f := f/(n+1);
   end if;
end fac;
```

Als VC ergibt sich

$[n \geq 0 \Rightarrow wp(IF, f = n!)]$

$\equiv$ If-Regel

$[n \geq 0 \Rightarrow (n = 0 \Rightarrow 1 = n!) \wedge (n \neq 0 \Rightarrow wp(\, fac(n+1, f); f := f/(n+1), f = n!))]$

$\equiv$ case analysis

$[(n \geq 0 \wedge n = 0 \Rightarrow 1 = n!) \wedge$
$(n \geq 0 \wedge n \neq 0 \Rightarrow wp(\, fac(n+1, f); f := f/(n+1), f = n!))]$

$\equiv$ Logik, Zuweisungsregel, Arithmetik

$[n \geq 0 \wedge n \neq 0 \Rightarrow wp(\, fac(n+1, f), f = (n+1)!)]$

$\Leftarrow$ Logik, Induktionsvoraussetzung, *vcb*

$[n \geq 1 \Rightarrow n+1 \geq 0 \wedge (\forall f : (n+1 \geq 0 \Rightarrow f = (n+1)!) \Rightarrow f = (n+1)!)]$

$\equiv$ Logik

true

Dieses Beispiel zeigt, daß Beweisregeln zum Beweis rekursiver Prozeduren ohne Terminierungsbeweis nicht zulässig sind, da mit ihnen auch nicht korrekte Prozeduren bewiesen werden können.

•

In [D92: 104-109] wird zum Beweis der Terminierung wie bei Schleifen eine Terminierungsfunktion *t* verwendet. Diese ist nach unten beschränkt und wird durch jeden einzelnen rekursiven Aufruf im Prozedurrumpf kleiner. Die Beweisregel lautet:

$[P \Rightarrow t \geq 0]$
$[P \wedge t = T \Rightarrow wp(body, R)]$ unter Verwendung der Voraussetzung
$[t^{x,y,z}_{a,b,c} < T \wedge P^{x,y,z}_{a,b,c} \Rightarrow wp(\, proc(a,b,c), R^{x,y,z}_{a,b,c})]$

für jeden rekursiven Aufruf von `proc` innerhalb des Prozedurrumpfes mit den jeweiligen aktuellen Parametern *a, b, c.*
Diese Beweisregel hat jedoch den Nachteil, daß man beim Beweis die Zusicherungen immer auf die in der Regel geforderte Form bringen muß, was nicht rein mechanisch machbar ist oder gar nicht möglich ist, wie das folgende Beispiel zeigt:

```
procedure add(x: in out integer; y: in natural) is
-- P: x = c ∧ y ≥ 0
-- R: x = y + c
-- t: y
   h: positive;
   k: integer;
begin
   h := y+1;
   k := x-1;
   if h /= 1 then
         add(k, h-2);
         x := x+1;
         x := k+2;
   end if;
end add;
```

Als VC ergibt sich

$[x = c \wedge y \geq 0 \Rightarrow y \geq 0] \equiv true$

$[x = c \wedge T = y \geq 0 \Rightarrow wp(body, x = y + c)]$

≡ If-Regel

$[x = c \wedge T = y \geq 0 \Rightarrow wp(h, k := y+1, x-1,$
$(h \neq 1 \Rightarrow wp(add(k, h-2), k+2 = y+c))$
$\wedge (h = 1 \Rightarrow x = y + c))]$

≡ Zuweisung

$[x = c \wedge T = y \geq 0 \Rightarrow (h \neq 1 \Rightarrow wp(add(k, h-2), k+2 = y+c))^{h,k}_{y+1,x-1} \wedge (y+1 = 1 \Rightarrow x = y+c))]$

≡ Logik

$[x = c \wedge T = y \geq 0 \wedge h \neq 1 \Rightarrow (wp(add(k, h-2), k+2 = y+c))^{h,k}_{y+1,x-1}]$

Induktionsvoraussetzung ist

$$[h-2 < T \wedge k = c \wedge h \geq 2 \Rightarrow wp(add(k, h-2), k = h-2+c)]$$

es ist also zu zeigen

$$[(h-2 < T \wedge k = c \wedge h \geq 2 \Rightarrow wp(add(k, h-2), k = h-2+c)) \Rightarrow$$
$$(x = c \wedge T = y \geq 1 \wedge h \neq 1 \Rightarrow (wp(add(k, h-2), k+2 = y+c))^{h,k}_{y+1,x-1})]$$

≡ Logik

$$[(h-2 < T \wedge k = c \wedge h \geq 2 \Rightarrow wp(add(k, h-2), k = h-2+c)) \wedge x = c \wedge T = y \geq 1 \wedge h \neq 1 \Rightarrow$$
$$(wp(add(k, h-2), k+2 = y+c))^{h,k}_{y+1,x-1}]$$

≡ ????

An dieser Stelle kann der Ausdruck nicht weiter umgeformt werden. Die Substitutionen können nicht durchgeführt werden, da die Voraussetzungen des Substitutionslemmas für *wp* nicht erfüllt sind.

•

Daher wird die Beweisregel von Bijlsma um den Beweis der Terminierung wie in [D92: 104-109] erweitert, was zu folgender Beweisregel führt:

Die Prozedur **procedure** proc(x: **in**; y: **in out**; z: **out**) ist spezifiziert durch (*P*, *R*) und die Terminierungsfunktion *t*. Dann muß gezeigt werden:

$[P \Rightarrow t \geq 0]$

$[P \wedge t = T \Rightarrow wp(body, R)]$, (*T* ist eine neue Variable)

wobei $wp(proc(a,b,c), N)$ des rekursiven Aufrufs im Prozedurrumpf *body* durch das stärkere Prädikat $vcb \wedge t^{x,y,z}_{a,b,c} < T$ approximiert wird (*vcbt* im Folgenden).

Da es vorkommen kann, daß in-out-Parameter und Spezifikationsvariablen wegen der Beweisregel von Bijlsma im Laufe des Beweises in einer Formel sowohl frei als auch gebunden vorkommen können, werden diese Variablen (aus Sicht der Formel sind Parameter und Spezifikationsvariablen syntaktisch Variablen) durch Anfügen eines Apostrophs (‘) umbenannt. Das wird durch die Beispiele 2 und 3 erläutert. Damit ergibt sich

$[P \Rightarrow t \geq 0]$

$[P^{y,m}_{y',m'} \wedge T = t^{y,m}_{y',m'} \Rightarrow wp(body^{y}_{y'}, R^{y,m}_{y',m'})]$, (*m* sind Spezifikationsvariablen)

Beispiele:

Beispiel 1: Fakultät:

```
procedure fac(n: in natural; f: out positive) is
-- P: n ≥ 0
-- R: f = n!
-- t: n
begin
   if n = 0 then f := 1;
   else fac(n-1, f);
        f := n * f;
   end if;
end fac;
```

Als VCs erhält man

1. $[n \geq 0 \Rightarrow n \geq 0] \equiv true$
2. $wp(body, f = n!)$

   $\Leftarrow vcbt$

   $T > n-1 \geq 0 \wedge (\forall f : (n-1 \geq 0 \Rightarrow f = (n-1)!) \Rightarrow f = (n-1)!$

zu zeigen ist

$[n \geq 0 \wedge n = T \Rightarrow (n \neq 0 \Rightarrow T > n-1 \geq 0 \wedge (\forall f : (n-1 \geq 0 \Rightarrow f = (n-1)!) \Rightarrow f = (n-1)!)]$

$\equiv$

true

•

Das folgende Beispiel ist die nichtterminierende Prozedur von oben:

```
procedure fac(n: in natural; f: out positive) is
-- P: n >= 0
-- R: f = n!
-- t: n
begin
   if n = 0 then f := 1;
   else  fac(n+1, f);
         f := f/(n+1);
   end if;
end fac;
```

Als VCs erhält man

1. $[n \geq 0 \Rightarrow n \geq 0] \equiv true$
2.

$$wp(body, f = n!)$$

$$\Leftarrow vcbt$$

$$T > n+1 \geq 0 \wedge (\forall f : (n+1 \geq 0 \Rightarrow f = (n+1)!) \Rightarrow f = (n+1)!$$

zu zeigen ist

$$[n \geq 0 \wedge n = T \Rightarrow (n \neq 0 \Rightarrow T > n+1 \geq 0 \wedge (\forall f : (n+1 \geq 0 \Rightarrow f = (n+1)!) \Rightarrow f = (n+1)!)]$$

$$\equiv$$

false

•

Mit einer anderen Terminierungsfunktion kann Teil 2 gezeigt werden:

```
procedure fac(n: in natural; f: out positive) is
-- V: n >= 0
-- N: f = n!
-- t: -n
begin
   if n = 0 then f := 1;
   else  fac(n+1, f);
         f := f/(n+1);
   end if;
end fac;
```

Als 2. VC erhält man

2.

$$wp(body, f = n!)$$

$$\Leftarrow vcbt$$

$$T > -(n+1) \wedge n+1 \geq 0 \wedge (\forall f : (n+1 \geq 0 \Rightarrow f = (n+1)!) \Rightarrow f = (n+1)!$$

zu zeigen ist

$$[n \geq 0 \wedge -n = T \Rightarrow (n \neq 0 \Rightarrow T > -(n+1) \wedge n+1 \geq 0 \wedge (\forall f : (n+1 \geq 0 \Rightarrow f = (n+1)!) \Rightarrow f = (n+1)!)]$$

$$\equiv$$

true

Allerdings ist hier Bedingung 1, die Beschränktheit der Terminierungsfunktion nach unten, nicht erfüllt. Dieses Beispiel zeigt die Notwendigkeit dieser Bedingung.

•

Mit Typprüfung sieht das Beispiel so aus, wobei $n \in 0..NMAX$ (=: *ntyp*) und $f \in 1..2^{63}-1$ (=: *ftyp*):

```
procedure fac(n: in ntyp; f: out ftyp) is
-- P: n ∈ ntyp
-- R: f = n!
-- t: n
begin
   if n = 0 then f := 1;
   else fac(n-1, f);
        f := n * f;
   end if;
end fac;
```

Als VCs erhält man

1. $[n \in ntyp \Rightarrow n \geq 0] \equiv true$
2. $wp(body, f = n!)$

   $\Leftarrow vcbt$

   $T > n-1 \geq 0 \wedge (\forall f : (n-1 \in ntyp \Rightarrow f = (n-1)!) \Rightarrow f = (n-1)! \wedge nf \in ftyp$

zu zeigen ist

$[n \in ntyp \wedge n = T \Rightarrow$

$(n \neq 0 \Rightarrow T > n-1 \geq 0 \wedge (\forall f : (n-1 \in ntyp \Rightarrow f = (n-1)!) \Rightarrow f = (n-1)! \wedge nf \in ftyp)]$

$\equiv$

$$\begin{cases} true, & falls\ 0 \leq NMAX \leq 20 \\ false & sonst \end{cases}$$

•

Beispiel 2: Addition:
Dieses Beispiel illustriert die Verwendung von in-out-Parametern und Spezifikationsvariablen. In den Formeln des Korrektheitsbeweises sind das Variablen, die sowohl frei (in *P* und *R*) als auch gebunden (durch die Beweisregel) vorkommen können. Die freien Vorkommen werden deshalb durch Anfügen eines Apostrophs (‘) umbenannt.

```
procedure add(x: in out integer; y: in natural) is
-- P: x = c ∧ y ≥ 0
-- R: x = y + c
-- t: y
begin
   if y /= 0 then
      add(x, y-1);
      x := x+1;
   end if;
end add;
```

Als VCs erhält man

$[P \Rightarrow t \geq 0]$

$[P \wedge t = T \Rightarrow wp(body, R)]$

1. $[x' = c' \wedge y \geq 0 \Rightarrow y \geq 0] \equiv true$
2. $[x' = c' \wedge y \geq 0 \wedge y = T \Rightarrow wp(body^{x}_{x'}, x' = y + c')]$

$wp(body^{x}_{x'}, x' = y + c')$

$\Leftarrow$ *vcbt*

$y - 1 < T \wedge (\exists c : x' = c \wedge y - 1 \geq 0) \wedge$
$(\forall x : (\forall c : x' = c \wedge y - 1 \geq 0 \Rightarrow x = y - 1 + c) \Rightarrow x' + 1 = y - 1 + c') \vee y = 0 \wedge x' = y + c'$

$\equiv$ mit c = x', Logik

$(y \neq 0 \Rightarrow y - 1 < T \wedge y \geq 1 \wedge (\forall x : (y \geq 1 \Rightarrow x = y + x' - 1) \Rightarrow x' + 1 = y - 1 + c')) \wedge x' = c'$
$\Leftarrow$ zu zeigen (*)

$x' = c' \wedge y \geq 0 \wedge y = T$

(*) Substitution
$[y \geq 0 \Rightarrow (y \neq 0 \Rightarrow y - 1 < y \wedge y \geq 1 \wedge (\forall x : (y \geq 1 \Rightarrow x = y + x' - 1) \Rightarrow x + 1 = y + x')) \wedge x' = x']$
$\equiv$
true

•

Beispiel 3: Addition, komplizierter:

Das folgende Beispiel ist die rekursive Prozedur von oben, die mit der Regel von Dahl nicht bewiesen werden kann.

```
procedure add(x: in out integer; y: in natural) is
-- P: x = c ∧ y ≥ 0
-- R: x = y + c
-- t: y
   h: positive;
   k: integer;
begin
   h := y+1;
   k := x-1;
   if h /= 1 then
         add(k, h-2);
         x := x+1;
         x := k+2;
   end if;
end add;
```

Als VCs erhält man

$[P \Rightarrow t \geq 0]$
$[P \wedge t = T \Rightarrow wp(body, R)]$

1. $[x' = c' \wedge y \geq 0 \Rightarrow y \geq 0] \equiv true$
2. $[x' = c' \wedge y \geq 0 \wedge y = T \Rightarrow wp(body^{x}_{x'}, x' = y + c')]$

$wp(body^{x}_{x'}, x' = y + c')$

$\Leftarrow$ *vcbt*

$(y \neq 0 \Rightarrow y - 1 < T \wedge y \geq 1 \wedge (\forall x : (y \geq 1 \Rightarrow x = y + x' - 2) \Rightarrow x + 2 = y + c')) \wedge x' = c'$

$\Leftarrow$ zu zeigen (*)

$x' = c' \wedge y \geq 0 \wedge y = T$

(*) Substitution
$[y \geq 0 \Rightarrow (y \neq 0 \Rightarrow y - 1 < y \wedge y \geq 1 \wedge (\forall x : (y \geq 1 \Rightarrow x = y + x' - 2) \Rightarrow x + 2 = y + x')) \wedge x' = x']$
≡
true

•

Beispiel 4: McCarthy's 91-Funktion [M83]:
Das ist ein akademisches Standardbeispiel für eine rekursive Funktion, bei der man nicht durch bloßes „Hinsehen" erkennt, was die Funktion berechnet bzw. wie das zu beweisen ist. Die in der Literatur vorkommenden Beweise sind alle mehr oder weniger „trickreich". Hier wird ein rein mechanischer Beweis gezeigt.

```
procedure p(x: in integer; y: out integer) is
-- P: MININT <= x <= MAXINT
-- R: x > 100 ∧ y = x - 10 ∨ x ≤ 100 ∧ y = 91
-- t: MAXINT - x
q: integer;
begin
   if x > 100 then y := x - 10;
   else p(x + 11, q); p(q, y);
   end if;
end p;
```

Als VCs erhält man

$[MININT \leq x \leq MAXINT \Rightarrow MAXINT - x \geq 0] \equiv true$

$[t = T \wedge true \Rightarrow wp(body, x > 100 \wedge y = x - 10 \vee x \leq 100 \wedge y = 91)]$

$wp(body, x > 100 \wedge y = x - 10 \vee x \leq 100 \wedge y = 91)$

$\Leftarrow$ *vcbt*

$t_q^x < T \wedge (q \leq 100 \Rightarrow x \leq 101) \wedge (q > 100 \wedge x > 100 \Rightarrow q = x) \wedge (q > 100 \wedge x \leq 100 \Rightarrow q = 101)$

$wp(p(x + 11, q), wp(p(q, y), x > 100 \wedge y = x - 10 \vee x \leq 100 \wedge y = 91))$

$\Leftarrow$ *vcbt*

$t_{x+11}^x < T \wedge (x > 89 \Rightarrow t_{x+1}^x < T) \wedge (x \leq 89 \Rightarrow t_{91}^x < T) \wedge x \leq 100$

zu zeigen:
$[t = T \wedge x \leq 100 \Rightarrow t_{x+11}^x < T \wedge (x > 89 \Rightarrow t_{x+1}^x < T) \wedge (x \leq 89 \Rightarrow t_{91}^x < T) \wedge x \leq 100]$
≡
true

•

Beispiel 5: Indirekte Rekursion:
Für indirekt rekursive Prozeduren arbeitet das Verfahren genauso. Um die Korrektheit einer der Prozeduren zu zeigen, muß die Korrektheit jeder verwendeten Prozedur gezeigt werden.

Beispiel:

```
procedure even(n: in natural; b: out boolean) is
-- P: n >= 0
-- R: b = (n mod 2 = 0)
-- t: n
begin
   if n = 0 then b := true;
```

```
    else odd(n-1, b);
    end if;
end even;

procedure odd(n: in natural; b: out boolean) is
-- P: n >= 0
-- R: b = (n mod 2 = 1)
-- t: n
begin
    if n = 0 then b := false;
    else even(n-1, b);
    end if;
end odd;
```

Korrektheit von even

$[n \geq 0 \Rightarrow n \geq 0] \equiv true$

$[T = n \geq 0 \Rightarrow (n = 0 \Rightarrow n \bmod 2 = 0) \wedge (n \neq 0 \Rightarrow wp(odd(n-1, b), b = (n \bmod 2 = 0)))]$

$\Leftarrow vcbt$

$[T = n \geq 1 \Rightarrow n-1 < T \wedge n-1 \geq 0 \wedge (\forall b : (n-1 \geq 0 \Rightarrow b = ((n-1) \bmod 2 = 1)) \Rightarrow b = (n \bmod 2 = 0))]$

$\equiv$

true

Korrektheit von odd (analog)

$[n \geq 0 \Rightarrow n \geq 0] \equiv true$

$T = n \geq 0 \Rightarrow (n = 0 \Rightarrow n \bmod 2 \neq 1) \wedge (n \neq 0 \Rightarrow wp(even(n-1, b), b = (n \bmod 2 = 1)))$

$\Leftarrow vcbt$

true

4 Beweis von Schleifen ohne Invariante

Dieses Kapitel stellt neue Techniken vor, mit denen unter bestimmten Bedingungen die Korrektheit von Schleifen gezeigt werden kann, die nur durch ihre Vor- und Nachbedingung spezifiziert sind, d.h. die ohne Invariante und Terminierungsfunktion gegeben sind. Ziel dabei ist, VCs für Schleifen genauso wie für andere Konstrukte zu generieren, bei denen die VC nur aus den Anweisungen und der Spezifikation generiert wird. Dabei ist das Vorgehen für For- und While-Schleifen sehr unterschiedlich. Bei For-Schleifen wird versucht, eine Invariante zu erzeugen und damit die Korrektheit zu zeigen. Zu diesem Zweck wird versucht, die Heuristik *Ersetzen einer Konstante durch eine Variable* zu automatisieren (Abschnitt 4.1). Bei While-Schleifen wird nicht versucht, eine Invariante zu erzeugen, sondern gleich die schwächste Vorbedingung (Abschnitt 4.2) berechnet.

4.1 Erzeugung von Invarianten für For-Schleifen

Zwei in der Literatur z.B. [G81: 195-205] vorkommende Heuristiken zum Finden einer Invarianten sind *Entfernung eines Konjunkts aus der Nachbedingung* und *Ersetzen einer Konstanten durch eine Variable*. Da es sich um Heuristiken handelt, sind diese Methoden nur schwer zu automatisieren. Doch bei For-Schleifen funktioniert sehr oft die zweite Heuristik, indem man die Schleifenobergrenze, die innerhalb der Schleife eine Konstante ist, durch die Schleifenvariable ersetzt.
Damit das automatisierbar ist, muß die Schleifenobergrenze eine Variable sein, denn nur Variablen sind in Ausdrücken ersetzbar. (Eine Variable im Sinne der Prädikatenlogik, d.h. auch ein als Konstante definierter Bezeichner ist hier Variable; eine Konstante im Sinne der Prädikatenlogik ist ein Zahlenliteral.) Man ersetzt einfach die Schleifenobergrenze in der Nachbedingung durch die Schleifenvariable und testet, ob das so entstandene Prädikat eine Invariante für die For-Schleife ist. Falls ja, ist die Schleife korrekt, falls nicht, ist das keine Invariante. Das kann 2 Gründe haben: die Schleife ist nicht korrekt, und es gibt gar keine Invariante (Satz 3.27); oder die Schleife ist korrekt, und das Verfahren kann keine Invariante ermitteln. In diesem Fall muß eine Invariante angegeben werden.
Falls Vor- und Nachbedingung als Konjunktion vorliegen und beide Zusicherungen gleiche gemeinsame Konjunkte haben, deren Variablen nicht im Rumpf *S* verändert werden, so sind diese gemeinsamen Konjunkte invariant. Das kommt besonders bei geschachtelten Schleifen vor. Es kann vorkommen, daß diese Konjunkte die Schleifenobergrenze enthalten. Wenn diese dann durch die Schleifenvariable ersetzt wird, hat man im Allgemeinen keine Schleifeninvariante mehr. Man darf die Ersetzung also nur in den Konjunkten der Nachbedingung vornehmen, die nicht in der Vorbedingung vorkommen.
Diese Methode funktioniert also höchstens, wenn die Obergrenze *b* eine Variable ist. Falls die Obergrenze keine Variable ist, könnte man auf die Idee kommen, vor Ausführung der Schleife `h := b;` einzuführen, und als Obergrenze dann die neue Hilfsvariable *h* zu verwenden. Das funktioniert im Allgemeinen aber nicht, da *h* in der Nachbedingung nicht vorkommt. Ersetzen einer nicht vorkommenden Variablen ergibt denselben Ausdruck, so daß man die Nachbedingung selbst als Invariante benutzen müßte, was in den meisten Fällen nicht geht.
Eine Lösung dieses Problems ist eine semantisch äquivalente Schleife, die als Obergrenze eine Variable hat. Diese Schleife wird durch eine Verschiebung *t* des Laufbereichs der Schleifenvariablen erzeugt, so daß die beiden For-Schleifen

```
-- V
for i in a .. b loop
  S(i);
end loop;
-- N
```

und

```
-- V
for i in t(a) .. t(b) loop
  S(t'(i));
end loop;
-- N
```

semantisch äquivalent sind. *t* ist eine Funktion auf der Menge *T* der Terme der Ganzzahlarithmetik $t : T \mapsto T$, und *t*‘ ist die zu *t* inverse Funktion (*T* ist wie üblich definiert). Die genaue Definition von *t* wird in den Definitionen 4.3 und 4.4 angegeben. Im Folgenden wird schrittweise auf diese Definition hingearbeitet , um zu verdeutlichen, wie sie zustande kommt. *t* muß folgende Eigenschaften erfüllen:

1. *t* ist auf $a..b$ definiert und bijektiv, damit die inverse Funktion existiert
2. $t(b) - t(a) = b - a$, d.h. die Anzahl der Schleifendurchläufe wird nicht verändert
3. $(\forall i, j : a \le i < j \le b : t(i) < t(j))$; *t* ist streng monoton steigend, damit die Reihenfolge der Schleifendurchläufe nicht verändert wird
4. $t(b)$ ist eine Variable
5. $t(b)$ kommt in *N* und *b* vor $(t(b) \in free(N) \cap free(b))$
6. *t* enthält keine Variablen, die in der Schleife verändert werden $((\forall v \in free(t) : trans(S, v)))$

Die ersten 3 Forderungen müssen erfüllt sein, damit die transformierte Schleife zur ursprünglichen semantisch äquivalent ist. Die 6. Forderung ist wichtig für den Fall, daß eine Variable im Schleifenrumpf zum ersten Mal einen Wert zugewiesen bekommt und somit vorher undefiniert ist. Dann würde *t* einen undefinierten Wert erhalten und Forderung 1 nicht erfüllt sein. Forderung 6 ist also eine Konsequenz aus Forderung 1, womit diese beiden Forderungen nicht unabhängig sind. Bei der Aufstellung der Forderungen wurde auch nicht auf Unabhängigkeit Wert gelegt, sondern auf Verständnis für den Grund der Forderungen.
Die Forderungen 4 und 5 sind für die Erzeugung der Invarianten notwendig.

Satz 4.1: Die Eigenschaften 1., 2. und 3. sind erfüllt, wenn gilt:

$$(\forall i : a \le i < b : t(i+1) - t(i) = 1)\ .$$

Beweis:

Eigenschaft 2: $a \le b$: sei $a + i = b \wedge i \ge 0$, Induktion über *i*:

$i = 0$: $t(b) - t(a) = t(a + i) - t(a) = t(a) - t(a) = 0$

$i \to i + 1$: Induktionsvoraussetzung $t(a + i) - t(a) = i$
zu zeigen: $t(a + i + 1) - t(a) = i + 1$

$t(a + i + 1) - t(a) = 1 + t(a + i) - t(a) = 1 + i$
(nach Voraussetzung des Satzes $(\forall i : a \le i < b : t(i+1) - t(i) = 1)$ und Induktionsvoraussetzung)

Falls $a > b$, gilt die Aussage wegen Allquantifizierung über den leeren Bereich.

Eigenschaft 1: *t* ist auf $a..b$ definiert: wenn *t* nicht auf $a..b$ definiert wäre, wäre mindestens ein Teilausdruck von $(\forall i : a \le i < b : t(i+1) - t(i) = 1)$ nicht definiert, und somit würde die Voraussetzung des Satzes nicht gelten.
t ist bijektiv, d.h. injektiv und surjektiv:

- *t* ist injektiv: zu zeigen $(\forall i, j : a \le i, j \le b \wedge i \ne j : t(i) \ne t(j))$
  angenommen, das gilt nicht, d.h. $(\exists i, j : a \le i, j \le b \wedge i \ne j : t(i) = t(j))$ und somit $t(i) - t(j) = 0$. Nach Eigenschaft 2 ist aber $t(i) - t(j) = i - j \ne 0$ für $i \ne j$.

- *t* ist surjektiv: da *t* injektiv ist und der Definitionsbereich von *t* diskret und endlich ist, folgt mit dem Schubfachprinzip, daß *t* auch surjektiv und damit bijektiv ist.

Eigenschaft 3: *t* ist streng monoton steigend, d.h. $(\forall i, j{:}a \le i < j \le b{:}t(i) < t(j))$

Sei $i + k = j \wedge k > 0$. Dann ist wegen Eigenschaft 2

$t(j) - t(i) = t(i+k) - t(i) = i + k - i = k > 0$, d.h. $t(i) < t(j)$.

•

Satz 4.2: Die Schleifen

```
for i in a .. b loop
  S(i);
end loop;
```

und

```
for i in t(a) .. t(b) loop
  S(t'(i));
end loop;
```

wobei *t* eine Funktion mit den Eigenschaften 1, 2 und 3 ist, sind semantisch äquivalent.

Beweis: die operationelle Semantik der 1. Schleife ist

```
S(a); S(a+1); S(a+2); ... ; S(b-1); S(b);
```

die operationelle Semantik der 2. Schleife ist

```
S(t'(t(a))); S(t'(t(a)+1)); S(t'(t(a)+2)); ... ; S(t'(t(b)-1)); S(t'(t(b)))
```

≡ Eigenschaften von *t*

```
S(t'(t(a))); S(t'(t(a+1))); S(t'(t(a+2))); ... ; S(t'(t(b-1))); S(b)
```

≡ Eigenschaften von *t*

```
S(a); S(a+1); S(a+2); ... ; S(b-1); S(b);
```

•

Da die Transformation nicht im Programm sondern nur mathematisch durchgeführt wird, kann es keine Bereichsüberschreitungen oder Seiteneffekte geben.
Wünschenswert ist auch, daß die Variable $t(b)$ nicht in *t* vorkommt. Diese Eigenschaft sowie die Voraussetzungen von Satz 4.1 sind von *N* unabhängig und erfüllt, wenn die Obergrenze *b* *geeignet* ist.

Definition 4.3: Ein Term ist *geeignet*, wenn er linear und keine Konstante ist und mindestens ein Koeffizient 1 ist, d. h. von der Form

$C + \sum_{j=1}^{n} c_j v_j$ ($n \ge 1, C, c_j \in \mathbb{Z}, c_k = 1$ für ein $k, 1 \le k \le n$, v_j verschiedene Variablen).

•

Dann ist *t* eine Funktion, die *b* nach v_k auflöst und folgendermaßen definiert ist:

Definition 4.4:

$$t(b) := b - C - \sum_{\substack{j=1 \\ j \ne k}}^{n} c_j v_j$$

$$t'(x) := x + C + \sum_{\substack{j=1 \\ j \ne k}}^{n} c_j v_j$$

•

Die Gründe für die Einschränkungen bezüglich der Form eines geeigneten Terms in Definition 4.3 sind:

1. $n = 0$ bedeutet, daß *b* eine Konstante ist. Ein Term, der eine Konstante ist, kann nicht nach einer Variablen aufgelöst werden. Man könnte eine Variable aus *N* addieren und *C* subtrahieren, aber das nützt im Allgemeinen nichts. Beispiel:

```
-- s = 0 ∧ n <= 10
for i in n .. 10 loop
  s := s + i;
end loop;
-- s = sum(j, n .. 10, j)
```

Mit $t(b) = b - 10 + n$ erhält man die semantisch äquivalente Schleife (bis auf eventuelle Bereichsüberschreitungen, die aber nicht vorkommen können)

```
-- s = 0 ∧ n <= 10
for i in 2n-10 .. n loop
  s := s+i+10-n;
end loop;
-- s = sum(j, n .. 10, j)
```

Als Invariante erhielte man $s = \sum_{j=i}^{10} j$, was nicht stimmt.

2. Wenn ein $c_k = 1$ ist, dann ergibt die Funktion $t(b) = v_k$ und v_k kommt nicht in *t* vor. *t* erfüllt dann die Voraussetzung von Satz 4.1.
Falls kein $c_k = 1$, dann gibt es keine Funktion *t* mit $t(b) = v_k$, ohne daß v_k in *t* vorkommt. Wenn z.B. $b = 2m$ ist, dann erreicht man *m* nur durch Subtraktion von *m*. Damit kommt *m* in *t* vor. Division durch 2 würde die Voraussetzung von Satz 4.1 verletzen.

3. Wenn zwei gleiche Variablen v_i und v_j mit Koeffizient 1 vorkommen, so ist zwar rein syntaktisch Punkt 2 erfüllt, aber es gibt es keine Funktion *t* mit $t(b) = v_k$, ohne daß v_k in *t* vorkommt; ein Beispiel ist $m + m$.

•

Forderung 5 lautet $t(b) \in free(N) \cap free(b)$. Falls $free(N) \cap free(b) = \varnothing$, ist keine Transformation der Schleife möglich (Beispiel 4). Falls $free(N) \cap free(b) \neq \varnothing$, gibt es mehrere mögliche Funktionen.
Sei $free(N) \cap free(b) = \{v_1, \ldots, v_n\}, n \geq 1$, dann gibt es *n* Funktionen $t_i, i = 1..n$ mit $t_i(b) = v_i$. Jede beliebige Teilmenge dieser *n* Funktionen kann zu einer Invarianten führen durch Ersetzung von v_i durch die Schleifenvariable in *N*. Weiter unten werden Beispiele mit $n = 2$ für jeden Fall angegeben, d.h. keine Funktion (Beispiel 5) bzw. genau eine (Beispiel 3) bzw. beide (Beispiel 2) führen zu einer Invarianten.
Die Verifikationsbedingungen werden aus denen für die For-Schleife mit gegebener Invariante (Abschnitt3.4.2) hergeleitet. Folgende For-Schleife sei gegeben:

```
-- V ∧ G
for i in a .. b loop
  S(i)
end loop;
-- N ∧ G
```

G ist das gemeinsame Konjunkt von Vor- und Nachbedingung, wie auf S.97 erwähnt. $t(b)$ ist eine Variable in *b* und *N*. Die transformierte Schleife lautet dann

```
-- V ∧ G
for i in t(a) .. t(b) loop
   S(t'(i))
end loop;
-- N ∧ G
```

Die mögliche Invariante ist dann $N_i^{t(b)} \wedge G$. Die Verifikationsbedingungen lauten

(1)

$[V \wedge G \wedge t(a) > t(b) \Rightarrow N \wedge G]$

$\equiv$ *t* streng monoton steigend, Logik

$[V \wedge G \wedge a > b \Rightarrow N]$

(2)

$[V \wedge G \wedge t(a) \leq t(b) \Rightarrow wp(S(t'(t(a))), (N_i^{t(b)} \wedge G)_{t(a)}^{i})]$

$\equiv$ *t* streng monoton steigend und bijektiv, Substitution, $i \notin Var(G)$

$[V \wedge G \wedge a \leq b \Rightarrow wp(S(a), N_{t(a)}^{t(b)} \wedge G)]$

$\equiv$ Konjunktivität von *wp*, Logik

$[V \wedge G \wedge a \leq b \Rightarrow wp(S(a), N_{t(a)}^{t(b)})]$

(3) $[t(a)+1 \leq i \leq t(b) \wedge N_{i-1}^{t(b)} \wedge G \Rightarrow wp(S(t'(i)), N_i^{t(b)})]$

(4)

$[t(a) \leq t(b) \wedge (N_i^{t(b)} \wedge G)_{t(b)}^{i} \Rightarrow N \wedge G]$

$\equiv$ Substitution, $i \notin Var(G)$

true

Die 4. VC ist schon auf Grund der Konstruktion erfüllt und entfällt somit bei diesem Verfahren. Falls die ersten 3 VCs bewiesen werden können, ist die Schleife korrekt. Falls nicht, ist das konstruierte Prädikat keine Invariante. Dann muß man die nächste Transformation ausprobieren (Beispiel 3). Da es sich bei dem Verfahren nur um eine Heuristik handelt, ist klar, daß das Verfahren nicht immer eine Invariante liefern kann, auch wenn die Schleife korrekt ist. Andererseits kann es vorkommen, daß eine Invariante auf eine andere Weise relativ einfach gefunden werden kann und von diesem Verfahren nicht erzeugt wird.
Die Transformation der Schleife ist möglich, wenn *b* ein geeigneter Term ist. Aber das ist keine wesentliche Einschränkung, da die Obergrenzen fast aller For-Schleifen diese Form haben [GB91]. Meistens sind die Obergrenzen von For-Schleifen Variablen, eine Summe einer Variablen und einer Konstanten oder die Summe von 2 Variablen. Die Form der Untergrenze spielt keine Rolle. Die komplizierteste Obergrenze in [GB91] ist ein Term der Form $\left\lfloor \frac{m}{2} \right\rfloor$.

Beispiele

Beispiel 1: Summation

```
-- s = 0
for i in 1 .. n loop
   s := s + i;
end loop;
-- s = sum(j, 1 .. n, j)
```

Obergrenze ist Variable, kein gemeinsames Konjunkt, keine Transformation erforderlich; einzige gemeinsame Variable ist *n*, Ersetzung von *n* durch *i* ergibt die Invariante $s = \sum_{j=1}^{i} j$.

•

Beispiel 2: Summation

```
-- s = 0
for i in 1 .. n + m loop
   s := s + i;
end loop;
-- s = sum(j, 1 .. n + m, j)
```

Obergrenze ist geeignet, kein gemeinsames Konjunkt, 2 gemeinsame Variablen, 2 Transformationen möglich

$t_1(x) = x - n, t_1'(x) = x + n$ oder
$t_2(x) = x - m, t_2'(x) = x + m$

beide Transformationen ergeben eine Invariante. Mit t_2 ergeben sich folgende VCs:

(1)

$$[s = 0 \wedge 1 > m + n \Rightarrow s = \sum_{j=1}^{m+n} j]$$

$\equiv$

true

(2)

$$[s = 0 \wedge 1 \leq m + n \Rightarrow wp(s := s + 1, s = \sum_{j=1}^{m+(1-m)} j)]$$

$\equiv$

true

(3)

$$[(1 - m) + 1 \leq i \leq n \wedge s = \sum_{j=1}^{m+i-1} j \Rightarrow wp(s := s + i + m, s = \sum_{j=1}^{m+i} j)]$$

$\equiv$

true

•

Beispiel 3: Summation

```
-- s = 0 ∧ n ≥ 0
for i in m .. m + n loop
   s := s + i;
end loop;
-- n ≥ 0 ∧ s = sum(j, 1 .. n, j) + m*(n+1)
```

Obergrenze ist geeignet, ein gemeinsames Konjunkt, 2 gemeinsame Variablen, 2 Transformationen möglich

$t_1(x) = x - n, t_1'(x) = x + n$ oder
$t_2(x) = x - m, t_2'(x) = x + m$

Die 1. Transformation führt zu $n \geq 0 \wedge s = (\sum_{j=1}^{n} j) + i \cdot (n + 1)$. Das ist keine Invariante der transformierten Schleife.

Die 2. Transformation führt zu $n \geq 0 \wedge s = (\sum_{j=1}^{i} j) + m \cdot (i+1)$. Nur das *n* in den nicht gemeinsamen Konjunkten darf ersetzt werden. Das ist eine Invariante der transformierten Schleife. Die VCs sind:

(1) $[n \geq 0 \wedge s = 0 \wedge m > m + n \Rightarrow s = (\sum_{j=1}^{n} j) + m \cdot (n+1)] \equiv true$

(2) $[n \geq 0 \wedge s = 0 \wedge m \leq m + n \Rightarrow wp(s := s + m, s = (\sum_{j=1}^{m-m} j) + m \cdot ((m-m)+1))] \equiv true$

(3)

$[m - m + 1 \leq i \leq n \wedge s = (\sum_{j=1}^{i-1} j) + m \cdot (i - 1 + 1) \Rightarrow wp(s := s + i + m, s = (\sum_{j=1}^{i} j) + m \cdot (i+1))]$

≡

true

•

Beispiel 4: Summation

```
-- s = 0 ∧ n > 1
for i in 1 .. n loop
  s := s + i;
end loop;
-- s > 2
```

Keine gemeinsamen Variablen in der Obergrenze und der Nachbedingung. Keine Transformation ist erforderlich, da die Obergrenze eine einfache Variable ist. Da diese nicht in der Nachbedingung vorkommt, ändert eine Ersetzung von *n* in *N* durch *i* nichts an *N*. Aber $s > 2$ ist keine Invariante.

•

Beispiel 5: Summation

```
-- s = 0 ∧ m ≥ 0 ∧ n ≥ 0
for i in 1 .. m + n loop
  s := s + i;
end loop;
-- s = sum(j, 1 .. n, j) + sum(j, n+1 .. m + n, j)
```

Die Obergrenze ist geeignet, 2 gemeinsame Variablen, kein gemeinsames Konjunkt, 2 Transformationen möglich:

$t_1(x) = x - m, t_1'(x) = x + m$ oder
$t_2(x) = x - n, t_2'(x) = x + n$

Keine der beiden Transformationen führt zu einer Invarianten der transformierten Schleife.

•

Beispiel 6: Bubble - Sort

Mit diesem etwas komplexeren Beispiel können mehrere Verfahren illustriert und erläutert werden. Es handelt sich um zwei geschachtelte Schleifen, bei der die Methode aus Abschnitt 3.4.4 verwendet wird. Daran sieht man erneut, daß man sich bei der Spezifikation und beim Beweis der inneren Schleife nur auf diese zu konzentrieren braucht, ohne die äußere Schleife zu berücksichtigen. D.h. die innere Schleife kann relativ unabhängig betrachtet werden.
Die hier vorgestellte Methode zum Beweis einer For-Schleife ohne gegebene Invariante funktioniert in dem Beispiel nicht für die äußere Schleife. Hier muß also eine Invariante angegeben werden. Die innere Schleife kann ohne gegebene Invariante bewiesen werden.
Außerdem kommt ein Prozeduraufruf mit Array-Parametern vor.

```
        -- Va: n ≥ 1 ∧ a = A
        for i in 1 .. n-1 loop
         -- a = perm(A) ∧ sort(a(n-i+2 .. n)) ∧ a(n-i+2) ≥ a(1 .. n-i+1)
         -- Vi: 1 ≤ i ≤ n-1
         for j in 1 .. n - i loop
               if a(j) > a(j+1) then swap(a(j), a(j+1)); end if;
         end loop;
         -- Ni: a(n-i+1) ≥ a(1 .. n-i+1)
        -- Pa: a = perm(A) ∧ sort(a(n-i+1 .. n)) ∧ a(n-i+1) ≥ a(1 .. n -
i)
        end loop;
        -- Na: a = perm(A) ∧ sort(a(1 .. n))
```

$a = perm(A)$ ≡ das Array *a* ist eine Permutation des Arrays *A*

$sort(a(n..m))$ ≡ das Array *a* ist von *n* bis *m* aufsteigend geordnet

$e \geq a(n..m)$ ≡ $e \geq$ jedes Element des Arrays *a* von *n* bis *m*

Korrektheit der äußere Schleife: Obergrenze ist geeignet, eine gemeinsame Variable, kein gemeinsames Konjunkt, eine mögliche Transformation:

$$t(x) = x + 1, t'(x) = x - 1$$

Das daraus resultierende Prädikat ist keine Invariante der inneren Schleife. Es besagt, daß die ersten *i* Elemente von *a* geordnet sind. Eine Invariante ist: die letzten *i* Elemente von *a* sind geordnet und größer oder gleich den ersten $n - i$ Elementen von *a*. Diese Invariante P_a muß angegeben werden.
Damit kann die Korrektheit der äußeren Schleife bewiesen werden. Hier erkennt man auch einen wichtigen Zusammenhang zwischen der Methode aus Abschnitt 3.4.4 und dem automatischen Erzeugen von Schleifeninvarianten. Denn nur durch Anwendung dieser Methode ist eine Invariante für die innere Schleife automatisch zu erzeugen. Ohne diese Methode müßten $[N_i \Rightarrow P_a]$ und $[(P_a)^i_{i-1} \Rightarrow V_i]$ gezeigt werden. Damit das funktioniert, müßten V_i und N_i so geändert werden, daß sie nicht mehr unabhängig von P_a sind. Das ist möglich, führt aber dazu, daß für die innere Schleife keine Invariante automatisch erzeugt werden kann.

Korrektheit der inneren Schleife:
Obergrenze ist geeignet, kein gemeinsames Konjunkt, 2 gemeinsame Variablen, 2 mögliche Transformationen

$$t_1(x) = x + 2i - n, t_1'(x) = x - 2i + n \text{ oder}$$
$$t_2(x) = x + i, t_2'(x) = x - i$$

t_1 führt zu einem Prädikat, das keine Invariante der inneren Schleife ist. Mit t_2 ergibt sich eine Invariante der inneren Schleife.

•

Für abwärts zählende For-Schleifen ergibt sich eine analoge Methode. Die VCs werden aus denen für die abwärtszählende For-Schleife mit gegebener Invariante (Abschnitt3.4.2.3) hergeleitet. Folgende For-Schleife sei gegeben:

```
-- V ∧ G
for i in reverse a .. b loop
  S(i)
end loop;
-- N ∧ G
```

G ist das gemeinsame Konjunkt von Vor- und Nachbedingung. $t_i(a)$ ist eine Variable in *a* und *N*. Die transformierte Schleife lautet dann

```
-- V ∧ G
for i in reverse t(a) .. t(b) loop
  S(t'(i))
end loop;
-- N ∧ G
```

Die mögliche Invariante ist dann $N_i^{t(a)} \wedge G$. Die Verifikationsbedingungen lauten

(1)

$[V \wedge G \wedge t(a) > t(b) \Rightarrow N \wedge G]$

$\equiv$ *t* streng monoton steigend, Logik

$[V \wedge G \wedge a > b \Rightarrow N]$

(2)

$[V \wedge G \wedge t(a) \leq t(b) \Rightarrow wp(S(t'(t(b))), (N_i^{t(a)} \wedge G)_{t(b)}^i)]$

$\equiv$ *t* streng monoton steigend und bijektiv, Substitution, $i \notin Var(G)$

$[V \wedge G \wedge a \leq b \Rightarrow wp(S(b), N_{t(b)}^{t(a)} \wedge G)]$

$\equiv$ Konjunktivität von *wp*, Logik

$[V \wedge G \wedge a \leq b \Rightarrow wp(S(b), N_{t(b)}^{t(a)})]$

(3) $[t(a) \leq i \leq t(b) - 1 \wedge N_{i+1}^{t(a)} \wedge G \Rightarrow wp(S(t'(i)), N_i^{t(a)})]$

(4) $[t(a) \leq t(b) \wedge (N_i^{t(a)} \wedge G)_{t(a)}^i \Rightarrow N]$

$\equiv$ Substitution, $i \notin Var(G)$

true

Beispiel:

Die innere For-Schleife eines Bubble-Sort-Algorithmus, bei dem das geordnete Teilarray nach jedem Durchlauf der äußeren Schleife von links nach rechts wächst, sieht so aus:

```
-- 1 ≤ i ≤ n
for j in reverse i+1 .. n loop
  if a(j) < a(j-1) then swap(a(j), a(j-1)); end if;
end loop;
-- a(i) ≤ a(i .. n)
```

$t(x) = x - 1, t'(x) = x + 1$

(1) $[V \wedge G \wedge a > b \Rightarrow N]$

$[1 \leq i \leq n \wedge i + 1 > n \Rightarrow a(i) \leq a(i..n)] \equiv true$

(2) $[V \wedge G \wedge a \leq b \Rightarrow wp(S(b), N_{t(b)}^{t(a)})]$

$[1 \leq i \leq n \wedge i + 1 \leq n \Rightarrow wp(if\ a(n) < a(n-1)\ then\ swap\ (a(n), a(n-1)), a(n-1) \leq a(n-1..n))]$

$\equiv$

true

(3) $[t(a) \leq i \leq t(b) - 1 \wedge N_{i+1}^{t(a)} \wedge G \Rightarrow wp(S(t'(i)), N_i^{t(a)})]$

$$[i \le j \le n-2 \wedge a(j+1) \le a(j+1..n) \Rightarrow$$
$$wp(\textit{if } a(j+1) < a(j) \textit{ then } swap(a(j+1), a(j)); \; a(j) \le a(j..n))]$$
$$\equiv$$
$$true$$

•

Einige zusammenfassende **Bemerkungen**:

1. Invarianten von For-Schleifen lassen sich manchmal relativ einfach aus der Nachbedingung ermitteln. Dazu muß die Obergrenze der For-Schleife ein geeigneter Ausdruck sein. Für abwärtszählende For-Schleifen sollte die Untergrenze der For-Schleife ein geeigneter Ausdruck sein. Falls beide Grenzen Konstanten sind, kann man die Schleife durch loop-unrolling verifizieren.
2. Es gibt For-Schleifen, bei denen es darauf ankommt, ob sie aufwärts- oder abwärtszählend sind, und es gibt For-Schleifen, wo das nicht der Fall ist. Im letzteren Fall wird dann meistens eine aufwärtszählende For-Schleife verwendet. Bei aufwärtszählenden For-Schleifen ist die Obergrenze meistens ein geeigneter Ausdruck. Bei abwärtszählenden For-Schleifen kommt es oft vor, daß die Untergrenze eine Konstante ist. Dann funktioniert diese Methode nicht.
3. Bei While-Schleifen funktioniert diese Methode im Allgemeinen nicht, da es keinen allgemeinen Zusammenhang zwischen den einzelnen Teilen gibt. Mit gegebener Terminierungsfunktion *t* kann man eine While- durch eine For-Schleife darstellen.

```
-- V
-- t
while B loop S end loop
-- N
```

Da $t > 0$ und mit jedem Schleifendurchlauf kleiner wird, d.h. um mindestens 1 abnimmt, ist die Anzahl der Schleifendurchläufe maximal *t*. Damit ist eine semantisch äquivalente For-Schleife

```
-- V
for i in 1 .. t loop
  if B then S end if;
end loop;
-- N
```

Dabei kann es vorkommen, daß *t* Variablen enthält, die in *S* geändert werden. Um das zu vermeiden, kann man in *t* Substitutionen der Form $\frac{v}{e}$ durchführen, wobei $v = e$ Teil der Vorbedingung ist. Wenn *e* keine Variable ist, ist das kein Problem. Falls *e* Variable ist, muß in *t* so substituiert werden, daß *t* möglichst keine Variable enthält, sie in *S* geändert wird und die in *V* vorkommt. Die letzte Forderung ist für die Anwendung der oben vorgestellten Methode nötig.

Als Beispiele werden die While-Schleifen von [G81] untersucht. Es stellt sich heraus, daß auf diese Weise genau die While-Schleifen verifiziert werden können, die „verkappte“ For-Schleifen sind. Das sind While-Schleifen, bei denen die genaue (und nicht nur die maximale) Anzahl der Durchläufe festliegt. Typischerweise sieht eine „verkappte“ For-Schleife so aus:

```
-- V ∧ n > 0 ∧ i = 1
while i < n loop
  S;
  i := i+1;
end loop;
-- N
```

Dabei gilt $trans(S, n) \wedge trans(S, i)$.

4.2 Berechnung der *wp* von While-Schleifen

Dieser Abschnitt beinhaltet eine neue Methode zur Berechnung der schwächsten Vorbedingung von While-Schleifen, die ohne Invariante und Terminierungsfunktion gegeben sind, d.h. es sind nur Vor- und Nachbedingung sowie die Schleife selbst gegeben. Damit ist es möglich, auch für While-Schleifen eine exakte VC anzugeben und damit so zu behandeln, wie die Anweisungen aus Abschnitt 3.2. Die Methode geht von dem Prädikat *wp für While-Schleifen* aus (Abschnitt 3.2.5). Nach [G81: 140] und [F89: 100] ist dieses Prädikat für praktische Anwendung wegen des großen Aufwands nicht geeignet. Doch gehen diese Aussagen von der Annahme aus, daß die *wp* von Hand berechnet wird. Es wird gezeigt, daß man dieses Prädikat doch dazu benutzen kann, mechanisch mit Hilfe des Computers eine While-Schleife zu verifizieren. Die Methode ist jedoch nur partiell in dem Sinne, daß sie nicht für alle Eingabeparameter ein Ergebnis liefert. Die Eingabe besteht aus einer While-Schleife *S* und einer Nachbedingung *N*, die Ausgabe ist entweder $wp(S, N)$ oder die Angabe, daß $wp(S, N)$ nicht ermittelt werden kann. Es ist kein syntaktisches Kriterium der Eingabedaten (S, N) bekannt, nach dem entschieden werden kann, ob $wp(S, N)$ ermittelt werden kann oder nicht.

4.2.1 Vorhandene Methoden zur Ermittlung einer Invarianten

Zunächst werden 3 Methoden aus [D92: 85-87] zur Ermittlung von Invarianten für While-Schleifen vorgestellt, von denen nur die 1. Methode funktioniert. Daß die beiden anderen Methoden nicht funktionieren, wird durch jeweils ein Gegenbeispiel gezeigt. Außerdem wird versucht herauszufinden, was an der verbalen Argumentation für die Korrektheit der Methoden nicht stimmt. Schließlich wird eine Modifikation von Methode 3 angegeben, mit der eine Invariante ermittelt werden kann.

In [D92: 85-87] werden 3 Methoden gezeigt, Invarianten zu finden. Sie basieren alle auf folgendem Vorgehen:

1. Schritt: Konstruiere eine Sequenz von Prädikaten P_0, P_1, P_2 ...
2. Schritt: Finde ein allgemeines Prädikat $P(i)$, bei dem sich durch Einsetzen von 0, 1, 2, ... für *i* jeweils P_0, P_1, P_2, ... ergibt
3. Schritt: Die Invariante ergibt sich aus Existenz- oder Allquantifizierung über $P(i)$

Wie sich zeigen wird, funktioniert nur die erste dieser 3 Methoden. Diese Verfahren sind teilweise automatisierbar, wobei sich hier Mensch und Maschine sehr gut ergänzen. Der 1. Schritt besteht im Wesentlichen in der Anwendung von *wp*- oder *sp*-Regeln. Die Vereinfachung der Prädikate erleichtert stark den 2. Schritt. Die Vereinfachung ist nur teilweise automatisierbar. Der 1. Schritt besteht in der Anwendung von *wp*- oder *sp*-Regeln (strongest postcondition [G81: 120], [DS90: 209-215]; *sp* ist schwieriger zu berechnen und liefert kompliziertere Ausdrücke. Da *sp* in dieser Arbeit nicht weiter benötigt wird, wird nicht weiter darauf eingegangen). Der 1. Schritt ist für den Computer ideal geeignet, da er rein mechanisch ist. Für den Menschen ist diese Aufgabe schlecht geeignet, da die Manipulation von komplizierten Ausdrücken sehr fehleranfällig und zeitaufwendig ist.
Der 2. Schritt beinhaltet das Erkennen von gewissen gemeinsamen Mustern und Unterschieden in einer Sequenz von Prädikaten. Dabei können die Gemeinsamkeiten und Unterschiede verschiedenster Natur sein. Es ist kein allgemeines Verfahren bekannt, diese automatisch zu erkennen. Allerdings ist es für den geübten Menschen oft möglich, sogar relativ einfach, diese Muster zu entdecken. Dieser Schritt ist für den Menschen einfach und für den Computer sehr schwer. Diese Aufgabe ist Gegenstand der Abschnitte 4.2.3-4.2.7.

Der 3. Schritt ist wieder leicht zu automatisieren. Es folgt ein Beispiel aus [G81: 147] zur Illustration dieser 3 Methoden. Der Algorithmus berechnet Quotient *q* und Rest *r* der Division von *x* durch *y*.

```
-- x ≥ 0 ∧ y > 0 ∧ q = 0 ∧ r = x
-- term: r
while r >= y loop
   r := r - y;
   q := q+1;
end loop;
-- 0 ≤ r < y ∧ q * y + r = x
```

Methode 1, die einzige funktionierende Methode:

Die Sequenz von Prädikaten wird folgendermaßen definiert:
P_0 ist die Vorbedingung. P_{i+1} steht mit P_i in der Relation

```
--Pi ∧ B
S
-- Pi+1
```

also $[P_i \wedge B \Rightarrow wp(S, P_{i+1})]$. Dann ist $(\exists i: i \geq 0: P_i)$ eine Invariante.

Beweis: Eine Invariante ist ein Prädikat, das die 5 Eigenschaften aus Abschnitt 3.4.1 erfüllt. Da hier keine Terminierungsfunktion und keine Nachbedingung betrachtet werden, kann der Zusammenhang zwischen diesen und der Invarianten nicht gezeigt werden. Man kann nur die folgenden 2 Eigenschaften zeigen:

1. Die Vorbedingung impliziert die Invariante:

   $[P_0 \Rightarrow (\exists i : i \geq 0 : P_i)]$

   $\equiv$ mit $i = 0$

   true

2. Die Invariante gilt immer am Anfang des Schleifenrumpfes:

   $[B \wedge (\exists i : i \geq 0 : P_i) \Rightarrow wp(S, (\exists i : i \geq 0 : P_i))]$

   $\equiv$ Skolemisierung, gebundene Umbenennung

   $[\forall i : i \geq 0 : B \wedge P_i \Rightarrow wp(S, (\exists j : j \geq 0 : P_j))]$

   $\equiv$ mit $j = i + 1$

   $[\forall i : i \geq 0 : B \wedge P_i \Rightarrow wp(S, P_{i+1})]$

   $\Leftarrow$ Definition P_{i+1}

   $[\forall i : i \geq 0 : B \wedge P_i \Rightarrow B \wedge P_i]$

   $\equiv$

   true

•

P_{i+1} kann mit Hilfe der Funktion *sp* induktiv berechnet werden:

$$P_{i+1} \equiv sp(P_i \wedge B, S)$$

Bemerkung: Die so berechnete Sequenz erfüllt die Relation $[P_i \wedge B \Rightarrow wp(S, P_{i+1})]$. Es kann aber auch jede andere Sequenz verwendet werden, die die Relation $[P_i \wedge B \Rightarrow wp(S, P_{i+1})]$ erfüllt, z.B. eine durch Probieren gefundene. Die Sequenz mit *sp* zu berechnen stellt aber ein Verfahren zur mechanischen Ermittlung einer solchen Sequenz dar.

•

Für das Beispiel ergibt sich die Sequenz

$$P_0 \equiv x \geq 0 \wedge y > 0 \wedge q = 0 \wedge r = x$$
$$P_1 \equiv x \geq 0 \wedge y > 0 \wedge q = 1 \wedge r + y = x \wedge r \geq 0$$
$$P_2 \equiv x \geq 0 \wedge y > 0 \wedge q = 2 \wedge r + 2y = x \wedge r \geq 0$$
$$P_3 \equiv x \geq 0 \wedge y > 0 \wedge q = 3 \wedge r + 3y = x \wedge r \geq 0$$
$$\vdots$$

Daraus erkennt man schon das allgemeine Prädikat

$$P(i) \equiv x \geq 0 \wedge y > 0 \wedge q = i \wedge r + iy = x \wedge r \geq 0$$

Das wird durch Einsetzen in die Rekursionsformel überprüft:

$$P_{i+1} \equiv sp(x \geq 0 \wedge y > 0 \wedge q = i \wedge r + iy = x \wedge r \geq 0 \wedge r \geq y, S)$$

≡ Zuweisungsregel für *sp*

$$x \geq 0 \wedge y > 0 \wedge q = i + 1 \wedge r + y + iy = x \wedge r + y \geq 0 \wedge r + y \geq y$$

≡ Arithmetik

$$x \geq 0 \wedge y > 0 \wedge q = i + 1 \wedge r + (i+1)y = x \wedge r + y \geq 0 \wedge r \geq 0$$

≡ Subsumierung

$$x \geq 0 \wedge y > 0 \wedge q = i + 1 \wedge r + (i+1)y = x \wedge r \geq 0$$

Damit ergibt sich als Invariante

$$(\exists i \geq 0 : x \geq 0 \wedge y > 0 \wedge q = i \wedge r + iy = x \wedge r \geq 0)$$

≡ Logik

$$x \geq 0 \wedge y > 0 \wedge r \geq 0 \wedge (\exists i \geq 0 : q = i \wedge r + iy = x)$$

≡ one-point rule

$$x \geq 0 \wedge y > 0 \wedge r \geq 0 \wedge r + qy = x$$

Daß dieses Prädikat tatsächlich eine Invariante ist, kann man leicht durch Einsetzen prüfen.

•

Methode 2 funktioniert nicht:

Q ist die Nachbedingung. Für Q_1 gilt

```
-- Q₁
S
-- ¬B ⇒ Q
```

Für Q_{i+1}, $i \geq 1$, gilt

```
-- Qᵢ₊₁
S
-- B ⇒ Qᵢ
```

Damit ist $Q_1 \equiv wp(S, \neg B \Rightarrow Q)$ und $Q_{i+1} \equiv wp(S, B \Rightarrow Q_i), i \geq 1$. Eine Invariante soll dann $(\forall i: i \geq 1: Q_i)$ sein. Für das Beispiel ergibt sich folgende Sequenz:

$$
\begin{aligned}
&Q \equiv 0 \leq r < y \wedge qy + r = x \\
&Q_1 \equiv r < 2y \Rightarrow y \leq r \wedge qy + r = x \\
&Q_2 \equiv 2y \leq r < 3y \Rightarrow qy + r = x \\
&Q_3 \equiv 3y \leq r < 4y \Rightarrow qy + r = x \\
&\qquad\qquad\quad \vdots
\end{aligned}
$$

Das allgemeine Prädikat ist

$$Q_i \equiv iy \leq r < (i+1)y \Rightarrow qy + r = x, i \geq 2$$

Überprüfen durch Einsetzen in die Rekursionsformel ergibt

$$
\begin{aligned}
&Q_{i+1} \equiv wp(S, r \geq y \Rightarrow (iy \leq r < (i+1)y \Rightarrow qy + r = x)) \\
&\equiv \\
&wp(S, r \geq y \wedge iy \leq r < (i+1)y \Rightarrow qy + r = x) \\
&\equiv \\
&wp(S, iy \leq r < (i+1)y \Rightarrow qy + r = x) \\
&\equiv \\
&iy \leq r - y < (i+1)y \Rightarrow (q+1)y + r - y = x \\
&\equiv \\
&(i+1)y \leq r < (i+2)y \Rightarrow qy + r = x
\end{aligned}
$$

Damit ergibt sich als angebliche Invariante

$$(\forall i \geq 2 : iy \leq r < (i+1)y \Rightarrow qy + r = x) \wedge (r < 2y \Rightarrow y \leq r \wedge qy + r = x)$$

≡ Logik

$$((\exists i \geq 2 : iy \leq r < (i+1)y) \Rightarrow qy + r = x) \wedge (r < 2y \Rightarrow y \leq r) \wedge (r < 2y \Rightarrow qy + r = x)$$

≡ Logik

$$(y > 0 \vee r < 2y \Rightarrow qy + r = x) \wedge (r < 2y \Rightarrow y \leq r)$$

Durch Einsetzen zeigt sich, daß dieses Prädikat keine Invariante ist. Tatsächlich funktioniert diese Methode nicht, wie das folgende einfachere Beispiel zeigt:

```
-- x = 0 ∨ x = 1
while x = 1 loop
      x := x-1;
end loop;
-- x = 0
```

Die Korrektheit dieser Schleife kann mit der Invarianten $x = 1 \vee x = 0$ und der Terminierungsfunktion x gezeigt werden. Folgende Sequenz erfüllt die Bedingungen für die Sequenz Q:

$$
\begin{aligned}
&Q \equiv x = 0 \\
&Q_1 \equiv x = 2 \vee x = 1 \\
&Q_2 \equiv true \\
&Q_3 \equiv true \\
&\quad \vdots
\end{aligned}
$$

Damit soll $x = 1 \vee x = 2$ eine Invariante ein, was aber nicht der Fall ist, wie man leicht nachprüfen kann.

In [D92: 85-87] befindet sich kein formaler Beweis für die Korrektheit dieser Methode, sondern nur eine verbale Argumentation. Ein Teil der Argumentationskette ist folgender: „Wenn die Schleife nach $i \geq 1$ Iterationen terminiert, dann war B i mal hintereinander *true* und danach *false*." Dieses Argument ist richtig. „In diesem Fall zeigen die Eigenschaften der Q_i, daß $B \wedge Q_i$ eine hinreichende Vorbedingung ist." Vorbedingung für was? Vermutlich dafür, daß B i mal hintereinander *true* und danach *false* ist. D.h. es soll o.B.d.A. für $i = 1$ gelten

```
-- Q₁ ∧ B
S
-- (¬B ⇒ Q) ∧ ¬B
```

Die Nachbedingung ist äquivalent zu $\neg B \wedge Q$. Das 1. Konjunkt der Nachbedingung gilt wegen der Konstruktion von Q_1. Doch das 2. Konjunkt gilt im Allgemeinen nicht. Wenn vor Ausführung von S $Q_1 \wedge B$ gilt, so gilt nach Ausführung von S nicht unbedingt $\neg B$.

Methode 3 funktioniert auch nicht:

Q ist die Nachbedingung. Für R_1 gilt

```
-- R₁
S
-- ¬B ∧ Q
```

Für R_{i+1}, $i \geq 1$, gilt

```
-- Rᵢ₊₁
S
-- B ∧ Rᵢ
```

Damit ist $R_1 \equiv wp(S, \neg B \wedge Q)$ und $R_{i+1} \equiv wp(S, B \wedge R_i), i \geq 1$. Eine Invariante soll dann $(\exists i : i \geq 1 : R_i)$ sein. Für das Beispiel ergibt sich folgende Sequenz:

$$
\begin{aligned}
R_1 &\equiv y \leq r < 2y \wedge qy + r = x \\
R_2 &\equiv 2y \leq r < 3y \wedge qy + r = x \\
R_3 &\equiv r \geq 2y \wedge 3y \leq r < 4y \wedge qy + r = x \\
&\vdots \\
R_i &\equiv (\forall j : 2 \leq j \leq i : jy \leq r) \wedge r < (i+1)y \wedge qy + r = x
\end{aligned}
$$

Als angebliche Invariante erhält man

$$(\exists i \geq 1 : y \geq 0 \wedge iy \leq r < (i+1)y) \wedge qy + r = x$$

$\equiv$ Logik

$$r \geq y \geq 0 \wedge qy + r = x$$

Das ist keine Invariante. Mit dem einfacheren Beispiel

```
-- x = 0 ∨ x = 1
while x = 1 loop
      x := x-1;
end loop;
-- x = 0
```

ergibt sich die Sequenz

$R_1 \equiv x = 1$
$R_2 \equiv x = 2$
$R_3 \equiv \mathit{false}$
$R_4 \equiv \mathit{false}$

Als angebliche Invariante erhält man $x = 1 \vee x = 2$, was aber nicht stimmt.

Die verbale Argumentation für diese Methode geht von der Gültigkeit der angeblichen Invarianten aus. Dann wird mit Hilfe der Eigenschaften von R_i eine korrekte Kette von Implikationen hergeleitet. Damit hat die verbale Argumentation die Form „Korrektheit der Methode $\Rightarrow$ *true*“. Das ist aber kein gültiges Vorgehen. Allerdings kann diese Methode leicht modifiziert werden, um eine korrekte Invariante zu erhalten. Die Konstruktion der R_i ergibt

```
:
    -- B ∧ R3
    S
    -- B ∧ R2
    S
    -- B ∧ R1
    S
    -- ¬B ∧ Q
```

Daraus ergibt sich die Invariante $B \wedge (\exists i \geq 1 : R_i) \vee \neg B \wedge Q$.

Beweis: Da hier keine Terminierungsfunktion und keine Vorbedingung betrachtet werden, kann der Zusammenhang zwischen diesen und der Invarianten nicht gezeigt werden. Man kann nur die folgenden 2 Eigenschaften zeigen:

1. Die Invariante und die Negation der Schleifenbedingung implizieren die Nachbedingung:

   $[(B \wedge (\exists i \geq 1 : R_i) \vee \neg B \wedge Q) \wedge \neg B \Rightarrow Q]$

   $\equiv$ Distributivität

   $[B \wedge (\exists i \geq 1 : R_i) \wedge \neg B \vee \neg B \wedge Q \wedge \neg B \Rightarrow Q]$

   $\equiv$ Logik

   $[\mathit{false} \vee \neg B \wedge Q \wedge \neg B \Rightarrow Q]$

   $\equiv$ Logik

   true

2. Die Invariante gilt immer am Anfang des Schleifenrumpfes:

   $[(B \wedge (\exists i \geq 1 : R_i) \vee \neg B \wedge Q) \wedge B \Rightarrow wp(S, B \wedge (\exists i \geq 1 : R_i) \vee \neg B \wedge Q)]$

   $\equiv$ Distributivität

   $[B \wedge (\exists i \geq 1 : R_i) \wedge B \vee \neg B \wedge Q \wedge B \Rightarrow wp(S, B \wedge (\exists i \geq 1 : R_i) \vee \neg B \wedge Q)]$

   $\equiv$ Logik

   $[B \wedge (\exists i \geq 1 : R_i) \vee \mathit{false} \Rightarrow wp(S, B \wedge (\exists i \geq 1 : R_i) \vee \neg B \wedge Q)]$

   $\Leftarrow$ Logik, schwache Disjunktivität

$[B \wedge (\exists i \geq 1 : R_i) \Rightarrow (\exists i \geq 1 : wp(S, B \wedge R_i)) \vee wp(S, \neg B \wedge Q)]$

≡ Skolemisierung, gebundene Umbenennung

$[(\forall i \geq 1 : B \wedge R_i \Rightarrow (\exists j \geq 1 : wp(S, B \wedge R_j)) \vee wp(S, \neg B \wedge Q))]$

≡ ∀-split

$[(\forall i \geq 2 : B \wedge R_i \Rightarrow (\exists j \geq 1 : wp(S, B \wedge R_j)) \vee wp(S, \neg B \wedge Q))] \wedge$
$[B \wedge R_1 \Rightarrow (\exists j \geq 1 : wp(S, B \wedge R_j)) \vee wp(S, \neg B \wedge Q)]$

≡ Definition R_i

$[(\forall i \geq 2 : B \wedge wp(S, B \wedge R_{i-1}) \Rightarrow (\exists j \geq 1 : wp(S, B \wedge R_j)) \vee wp(S, \neg B \wedge Q))] \wedge$
$[B \wedge wp(S, \neg B \wedge Q) \Rightarrow (\exists j \geq 1 : wp(S, B \wedge R_j)) \vee wp(S, \neg B \wedge Q)]$

≡ mit $j = i - 1$, Logik

$[(\forall i \geq 2 : B \wedge wp(S, B \wedge R_{i-1}) \Rightarrow i - 1 \geq 1 \wedge (wp(S, B \wedge R_{i-1}) \vee wp(S, \neg B \wedge Q)))] \wedge$
$true$

≡ Logik

$true$

•

Für das Beispiel der Division ergibt sich als Invariante $qy + r = x \wedge (r \geq x \geq y \vee 0 \leq r < y)$. Für das andere Beispiel ergibt sich als Invariante $x = 1 \vee x = 0$. Daß diese Prädikate tatsächlich Invarianten sind, kann man leicht durch Einsetzen prüfen.

Bei diesen Methoden ist es nötig, eine Terminierungsfunktion anzugeben. Die Invariante kann durch Betrachtung der Sequenz der erzeugten Prädikate (vielleicht) gefunden werden. Der größte Aufwand steckt im Allgemeinen im Finden eines allgemeinen Prädikates aus einer endlichen Anfangssequenz. Mit höchstens gleichem Aufwand ist es aber möglich, mehr zu erreichen, nämlich die Berechnung der schwächsten Vorbedingung einer Schleife $wp(LOOP, N)$.

4.2.2 Automatische Konstruktion von μf

Zur automatischen Konstruktion von μf, der schwächsten Vorbedingung einer While-Schleife (Abschnitt 3.2.5), wird versucht, das Vorgehen eines Menschlichen Bearbeiters zu mechanisieren. μf ist der kleinste Fixpunkt einer rekursiven Funktion. Die Ermittlung von μf erfolgt durch ein induktives Vorgehen. Zunächst werden einige Werte explizit ausgerechnet und aus diesen eine allgemeines Muster „erraten“. Bezogen auf das hier vorliegende Problem heißt das: zunächst wird rein mechanisch eine endliche Sequenz von Prädikaten erzeugt. Anschließend wird versucht, darin ein Muster zu erkennen. Dieses Erkennen ist auch die wesentliche Schwierigkeit, denn es ist nicht bekannt, wie ein menschlicher Bearbeiter die Mustererkennung durchführt. Falls ein Muster erkannt werden kann, kann ein allgemeines Prädikat berechnet werden. Zunächst wurde versucht, dieses Problem mit Hilfe von grammatical inference zu lösen, wie es am Ende dieses Abschnitts kurz dargestellt wird. Da sich dieser Weg als nicht praktisch erwiesen hat, wird versucht, dieses Problem mit Hilfe von Unifikation zu lösen (Abschnitte 4.2.3- 4.2.7).

Der kleinste Fixpunkt μf einer monotonen und stetigen Funktion *f* über einem vollständigen Verband wird durch sukzessive Anwendung von *f* auf das Infimum $\bot$ des Verbandes konstruiert, beginnend mit $\bot$. Das Infimum der Boolschen Algebra der Prädikate ist *false*. Die Schwierigkeit liegt darin, daß man nur endlich viele Anwendungen von *f* durchführen kann und man von vornherein nicht weiß, wie viele genügen. Die erzeugte Sequenz der Prädikate ist

$$P_0 \equiv false$$
$$P_1 \equiv f(false)$$
$$P_2 \equiv f^2(false)$$
$$\vdots$$

Dann ist $P_{\min\{i:P_i \equiv P_{i+1}\}}$ der kleinste Fixpunkt. D.h. wenn zwei aufeinanderfolgende Prädikate äquivalent sind, hat man den kleinsten Fixpunkt. Wenn die Anwendung von *f* aber immer nichtäquivalente Prädikate ergibt, heißt das nicht, daß es keinen kleinsten Fixpunkt gibt. Den gibt es ja wegen der Monotonie und Stetigkeit von *f*. In diesem Fall ist der kleinste Fixpunkt die kleinste obere Schranke aller P_i. D.h. für μf muß gelten:

$$[P_0 \Rightarrow \mu f]$$
$$[P_1 \Rightarrow \mu f]$$
$$[P_2 \Rightarrow \mu f]$$
$$\vdots$$

also

$$[\forall i : i \geq 0 : P_i \Rightarrow \mu f]$$
$$\equiv$$
$$[(\exists i : i \geq 0 : P_i) \Rightarrow \mu f]$$

$\mu f \equiv (\exists i : i \geq 0 : P_i)$ erfüllt diese Bedingung, und wegen $\equiv$ ist μf sogar das kleinste Prädikat, das diese Bedingung erfüllt. (Mit *kleinstes* ist hier das kleinste Element im Verband gemeint; auf Prädikate bezogen, bedeutet das *das stärkste Prädikat*). Daß μf tatsächlich ein Fixpunkt von *f* ist, muß noch gezeigt werden.

Satz 4.5: μf ist ein Fixpunkt von *f*.

Beweis: zu zeigen ist $f((\exists i: i \geq 0: P_i)) \equiv (\exists i: i \geq 0: P_i)$.

$$f((\exists i: i \geq 0: P_i))$$
$$\equiv \text{Definition von } P_i$$
$$f((\exists i: i \geq 0: f^i(false)))$$
$$\equiv \text{Definition von } f$$
$$\neg B \wedge N \vee B \wedge wp(S, (\exists i: i \geq 0: f^i(false)))$$
$$\equiv \text{starke Disjunktivität, Lemma 3.11}$$
$$\neg B \wedge N \vee B \wedge (\exists i: i \geq 0: wp(S, f^i(false)))$$
$$\equiv \text{Logik}$$
$$(\exists i: i \geq 0: \neg B \wedge N \vee B \wedge wp(S, f^i(false)))$$
$$\equiv \text{Definition von } f$$

$$(\exists i: i \geq 0: f\,(f^{\,i}(\mathit{false})))$$

$$\equiv$$

$$(\exists i: i \geq 0: f^{\,i+1}(\mathit{false}))$$

zu zeigen bleibt $(\exists i: i \geq 0: f^{\,i+1}(\mathit{false})) \equiv (\exists i: i \geq 0: f^{\,i}(\mathit{false}))$:

$\Rightarrow$:

$$(\exists i: i \geq 0: f^{\,i+1}(\mathit{false})) \Rightarrow (\exists i: i \geq 0: f^{\,i}(\mathit{false}))$$

$\equiv$ gebundene Umbenennung

$$(\exists i: i \geq 0: f^{\,i+1}(\mathit{false})) \Rightarrow (\exists j: j \geq 0: f^{\,j}(\mathit{false}))$$

$\equiv$ Skolemisierung

$$(\forall i: i \geq 0: f^{\,i+1}(\mathit{false}) \Rightarrow (\exists j: j \geq 0: f^{\,j}(\mathit{false})))$$

$\equiv$ mit $j = i + 1$

$$(\forall i: i \geq 0: f^{\,i+1}(\mathit{false}) \Rightarrow i + 1 \geq 0 \wedge f^{\,i+1}(\mathit{false}))$$

$\equiv$ Logik

$$\mathit{true}$$

$\Leftarrow$:

$$(\exists i: i \geq 0: f^{\,i}(\mathit{false})) \Rightarrow (\exists i: i \geq 0: f^{\,i+1}(\mathit{false}))$$

$\equiv$ gebundene Umbenennung

$$(\exists i: i \geq 0: f^{\,i}(\mathit{false})) \Rightarrow (\exists j: j \geq 0: f^{\,j+1}(\mathit{false}))$$

$\equiv$ Skolemisierung

$$(\forall i: i \geq 0: f^{\,i}(\mathit{false}) \Rightarrow (\exists j: j \geq 0: f^{\,j+1}(\mathit{false})))$$

$\equiv$ Logik, $\forall$-split

$$(\forall i: i \geq 1: f^{\,i}(\mathit{false}) \Rightarrow (\exists j: j \geq 0: f^{\,j+1}(\mathit{false}))) \wedge (f^{\,0}(\mathit{false}) \Rightarrow (\exists j: j \geq 0: f^{\,j+1}(\mathit{false})))$$

$\equiv$ mit $j = i - 1$

$$(\forall i: i \geq 1: f^{\,i}(\mathit{false}) \Rightarrow i - 1 \geq 0 \wedge f^{\,i}(\mathit{false})) \wedge (f^{\,0}(\mathit{false}) \Rightarrow (\exists j: j \geq 0: f^{\,j+1}(\mathit{false})))$$

$\equiv$ Logik

$$\mathit{true} \wedge (\mathit{false} \Rightarrow (\exists j: j \geq 0: f^{\,j+1}(\mathit{false})))$$

$\equiv$ Logik

$$\mathit{true} \wedge \mathit{true}$$

•

Die automatische Ermittlung des Prädikates $P(i)$ stellt das gleiche Problem wie im vorigen Abschnitt dar. Im Folgenden wird ein Verfahren zur partiellen Lösung dieses Problems vorgestellt. Damit ist gemeint, daß das Verfahren aus einer gegebenen endlichen Sequenz von Prädikaten ein allgemeines Prädikat ermitteln kann oder keines findet und erfolglos abbricht. Das Verfahren kann natürlich bei Weitem nicht alle Probleme dieser Art lösen, genau wie ein Mensch. Das bedeutet, daß es vorkommen kann, daß kein allgemeines Prädikat gefunden werden kann. Das Verfahren kann irren genau wie ein Mensch. Das bedeutet, daß ein allge-

meines Prädikat gefunden wird, das auf das betrachtete Anfangsstück paßt, aber nicht auf alle folgenden. Ein solcher Irrtum ist aber entscheidbar, indem versucht wird, festzustellen, ob das auf diese Weise gefundene Prädikat tatsächlich auf alle paßt. Das ist ein Induktionsbeweis. Formal sieht das *Problem des allgemeinen Prädikates* so aus:

> Gegeben ist eine endliche Sequenz von Prädikaten $P_i, i \geq 0$, die durch $P_i \equiv f^i(\mathit{false})$ definiert ist. Gesucht ist ein Prädikat $P(i)$, so daß $(\forall i: i \geq 0: P(i) \equiv P_i)$ und in $P(i)$ darf weder die Funktion *f* noch *wp* vorkommen.

Zur Lösung des Problems wird die endliche Anfangssequenz $P_0, P_1, \ldots, P_n$ betrachtet. Es ist klar, daß es unendlich viele Prädikate $P(i)$ mit $(\forall i: 0 \leq i \leq n: P(i) \equiv P_i)$ gibt. Z.B. ist $P(i) \equiv (\exists j: 0 \leq j \leq n: i = j \wedge P_j) \vee R(i)$ für beliebiges R so eine Menge von Prädikaten. Denn es gilt $P(i) \equiv \begin{cases} P_i, & \mathit{falls}\ i \leq n \\ R(i), & \mathit{falls}\ i > n \end{cases}$. Im Allgemeinen gilt aber nicht $(\forall i: i \geq 0: P(i) \equiv P_i)$.

Man muß vielmehr versuchen, in $P_0, P_1, \ldots, P_n$ ein gemeinsames Muster bzw. die Unterschiede zu entdecken.

Beispiel (ganzzahlige Division):

$$
\begin{aligned}
P_0 &\equiv \mathit{false} \\
P_1 &\equiv 0 \leq r < y \wedge qy + r = x \\
P_2 &\equiv 0 \leq r < 2y \wedge qy + r = x \\
P_3 &\equiv 0 \leq r < 3y \wedge qy + r = x \\
&\vdots
\end{aligned}
$$

Der Unterschied liegt im Koeffizienten von *y*. Man kann jetzt $P(i) \equiv 0 \leq r < iy \wedge qy + r = x$ raten. Um das zu verifizieren, wird ein Induktionsbeweis durchgeführt.

$i = 0$: $P(0) \equiv 0 \leq r < 0y \wedge qy + r = x \equiv \mathit{false}$

$i \to i+1$:

$P(i+1)$

$\equiv$ Definition von P

$f^{i+1}(\mathit{false})$

$\equiv$

$f(f^i(\mathit{false}))$

$\equiv$ Definition von P

$f(P(i))$

$\equiv$ Induktionsvoraussetzung

$f(0 \leq r < iy \wedge qy + r = x)$

$\equiv$ Definition von f

$0 \leq r < y \wedge qy + r = x \vee r \geq y \wedge wp(r := r - y; q := q + 1, 0 \leq r < iy \wedge qy + r = x)$

$\equiv$ Zuweisungsregel, Distributivität

$(0 \leq r < y \vee y \leq r < (i+1)y) \wedge qy + r = x$

$\equiv$ Logik

$0 \le r < (i+1)y \wedge qy + r = x$

•

Nach Satz 3.16 und Satz 4.5 ist die schwächste Vorbedingung die Disjunktion über die Menge der Prädikate $P(i)$. Damit ist

$$wp(LOOP, N) \equiv (\exists i \ge 0 : 0 \le r < iy) \wedge qy + r = x$$
$$\equiv$$
$$y \ge 0 \wedge r \ge 0 \wedge qy + r = x$$

Und es gilt $[x \ge 0 \wedge y > 0 \wedge q = 0 \wedge r = x \Rightarrow y \ge 0 \wedge r \ge 0 \wedge qy + r = x]$. Damit ist die Korrektheit bewiesen.

•

Dieses Problem, in einer Sequenz von Prädikaten ein allgemeines Muster zu finden, ähnelt entfernt dem Problem der *grammatical inference* [F82: 349-410]. Es wurden verschiedene *grammatical-inference*-Algorithmen ([F82: 367-370], [F82: 391-398] und [BCW90: 191-196]) daraufhin untersucht, ob sie zur Lösung des hier vorliegenden Problems geeignet sind. Da sich keiner dieser Algorithmen als geeignet herausstellte, wurde dieser Weg nicht weiterverfolgt.
Die Ermittlung der schwächsten Vorbedingung beruht im Wesentlichen in der Entdeckung des Musters bzw. der Unterschiede in der Sequenz der Prädikate. Die Prädikate in der Sequenz haben hier die gleiche Struktur, da sie sich nur in Konstanten unterscheiden. Jetzt stellt sich die Frage, was es genau heißt, eine Sequenz von Prädikaten ist strukturgleich und, damit verbunden, wie man diese Strukturgleichheit einerseits und die Unterschiede andererseits feststellt. Diese intuitive Vorstellung von Strukturgleichheit wird im Folgenden formalisiert. $S(P)$ heißt, eine Sequenz P von Prädikaten ist strukturgleich. Eine formale Definition von Strukturgleichheit wird zwar erst in Abschnitt 4.2.5 gegeben, doch im Folgenden wird schrittweise auf diese Definition hingearbeitet, wobei der Begriff *Strukturgleichheit* und $S(P)$ ohne formale Definition verwendet werden.

4.2.3 Klassifikation des Problems

Ein anderes Problem, das sich mit so etwas wie Strukturgleichheit von Ausdrücken befaßt, ist die Unifikation [S88]. Im nächstem Abschnitt wird gezeigt, wie man das Problem der Strukturgleichheit auf ein Unifikationsproblem reduzieren kann. Doch zunächst muß genau festgelegt werden, was mit Unifikation in diesem Zusammenhang überhaupt gemeint ist. $U(P)$ heißt , eine Sequenz *P* von Prädikaten ist unifizierbar. Da hier semantische Eigenschaften von Funktionssymbolen, wie z.B. Kommutativität, berücksichtigt werden müssen, ist im Folgenden mit Unifikation, unifizierbar, Unifikator u.ä. immer die sogenannte *E*-Unifikation [BS94] (equational unification) usw. im Gegensatz zur sogenannten freien oder syntaktischen Unifikation gemeint. Die semantischen Eigenschaften der Funktionssymbole werden durch Axiome, die Gleichungen (equations) sind, beschrieben. *E* bezeichnet dann die Vereinigung der Axiomenmengen der einzelnen nicht freien Funktionssymbole. Z.B. sind die Terme $f(x)+1$ und $1+y$ nicht frei unifizierbar, jedoch C_+-unifizierbar mit dem C_+-Unifikator $y \mapsto f(x)$, wobei C_+ commutativity bedeutet und die Axiomenmenge $\{a+b=b+a\}$ bezeichnet. In diesem Fall ist $E = C_+ \cup \varnothing_f$. In diesem Zusammenhang wichtige Axiome sind:

1. $C_f = \{f(a,b) = f(b,a)\}$ (Kommutativität)
2. $A_f = \{f(a, f(b,c)) = f(f(a,b),c)\}$ (Assoziativität)

3. $I_f = \{ f(a,a) = a \}$ (Idempotenz)
4. $U_f = \{ f(a,u) = a \}$ (Einselement)
5. $Z_f = \{ f(a,z) = z \}$ (Nullelement)

Mit diesen Axiomen ist $E \subseteq C \cup A \cup I \cup U \cup Z \wedge E \neq \varnothing$. Die *E*-Unifikation ist viel schwieriger, d.h. aufwendiger oder gar nicht lösbar (abhängig von *E*) als die freie Unifikation, die man als einen Spezialfall der *E*-Unifikation mit *E*=∅ betrachten kann. Für die freie Unifikation gibt es effiziente (mit linearem bzw. quadratischen Aufwand) Algorithmen von Paterson, Wegman [PW78] und Martelli, Montanari [MM82]. Die *E*-Unifikation kann in die folgenden 3 Klassen steigender Schwierigkeit eingeteilt werden:

1. Die *E*-Unifikation heißt *elementar*, falls alle Funktionssymbole die Eigenschaft *E* haben.
2. Die *E*-Unifikation heißt *mit Konstanten*, falls neben Funktionssymbolen mit der Eigenschaft *E* nur noch freie Konstanten, d.h. nullstellige Funktionssymbole, vorkommen.
3. Die *E*-Unifikation heißt *allgemein*, falls neben Funktionssymbolen mit der Eigenschaft *E* noch beliebig viele freie Funktionssymbole mit beliebiger Stelligkeit vorkommen.

Das Problem der Strukturgleichheit wird auf ein allgemeines *E*-Unifikationsproblem zurückgeführt, da neben Funktionssymbolen mit semantischen Eigenschaften auch beliebig viele freie Funktionssymbole vorkommen können, z.B. Array-Bezeichner. Ein weiters Problem ist, daß verschiedene Funktionssymbole verschiedene semantische Eigenschaften haben können. Z.B. ist + assoziativ und kommutativ (*AC*), und ∧ ist assoziativ, kommutativ und idempotent (*ACI*). Das führt zum Problem der Kombination verschiedener *E*-Unifikationsalgorithmen [BS94]:

> Gegeben sind *n* Algorithmen für E_1-Unifikation, ..., E_n-Unifikation. Wie kann man diese zu einem Algorithmus für die *E*-Unifikation mit $E = E_1 \cup \cdots \cup E_n$ kombinieren?

Dieses Problem wurde zunächst für Theorien gelöst, die *collapse free* und *regular* sind. Eine Theorie ist *collapse free*, wenn sie kein Axiom der Form $t = x$ enthält, wobei *t* ein echter Term ist, d.h. keine Variable, und *x* eine Variable. Die Axiome *C* und *A* sind collapse free, die Axiome *I*, *U* und *Z* nicht. Eine Theorie ist *regular*, wenn für alle ihre Axiome $lhs = rhs$ gilt: $free(lhs) = free(rhs)$. Die Axiome *C*, *A* und *I* sind regular, die Axiome *U* und *Z* nicht. In allgemeiner Form wurde das Problem zuerst von Schmidt-Schauß gelöst [S89]. Allerdings können mit diesem Verfahren nur die vollständigen Mengen aller allgemeinsten Unifikatoren berechnet werden, falls diese endlich sind. Eine andere und wesentliche bessere Lösung stammt von Baader und Schulz [BS92], mit der es sowohl möglich ist, einzelne Unifikatoren zu berechnen als auch die Unifizierbarkeit zu entscheiden. (Durch die Berechnung der vollständigen Mengen aller Unifikatoren kann man zwar auch einzelne Unifikatoren berechnen und die Unifizierbarkeit entscheiden, allerdings mit viel höherem Aufwand). Die hier nicht frei vorkommenden Funktionssymbole sind +, · und ≡ mit den Eigenschaften *AC*, sowie ∧, ∨, min und max mit den Eigenschaften *ACI*. Da die Assoziativität nur bei mindestens 3 Argumenten auftritt und = immer mit genau 2 Argumenten vorkommt, kann man = auch mit der Eigenschaft *AC* betrachten. Die Axiome *U* und *Z* brauchen nicht betrachtet zu werden, da diese durch den in Analytica bzw. Mathematica vorhandenen simplifier bei der Generierung der Prädikate sofort angewendet werden. Damit ergibt sich ein *E*-Unifikationsproblem mit

$$E = AC_{+} \cup AC_{*} \cup AC_{=} \cup AC_{\equiv} \cup ACI_{\wedge} \cup ACI_{\vee} \cup ACI_{\min} \cup ACI_{\max} \cup \varnothing_{Rest}.$$

Dabei beinhaltet *Rest* alle anderen Funktionssymbole, insbesondere auch die Relationssymbole und Quantoren, die rein syntaktisch als 3-stellige freie Funktionssymbole ohne Semantik betrachtet werden.
Da *AC*-Unifikation und *ACI*-Unifikation NP-vollständige Probleme sind, gilt das nach [BS94] auch für das hier vorliegende *E*-Unifikationsproblem. Die NP-Vollständigkeit eines Problems bedeutet noch nicht die praktische Unlösbarkeit des Problems. Das Verfahren von [BS92] wurde von [KR97] implementiert.

4.2.4 Überblick über das Verfahren

In diesem Abschnitt wird ein Überblick über das Verfahren in Form eines Algorithmus in Pseudo-Code gegeben. Dabei werden teilweise Begriffe und Funktionen verwendet, die erst in den folgenden Abschnitten formal definiert werden. Beispiele zur Erläuterung des Verfahrens befinden sich im letzten Abschnitt dieses Kapitels (Abschnitt 4.2.7.3)
Der folgende Algorithmus berechnet *wp* für eine While-Schleife und eine Nachbedingung oder gibt `no_success` zurück, falls das nicht möglich ist. Dabei wird Unifikation an 2 Stellen verwendet. Zunächst wird mit Hilfe der Unifikation festgestellt, wo eine Sequenz strukturgleicher Prädikate beginnt (Zusicherung A1). Falls es eine Sequenz strukturgleicher Prädikate gibt, werden daraus wieder mit Hilfe der Unifikation mehrere Zahlensequenzen extrahiert (Zusicherung A2). Zu jeder Zahlensequenz wird dann eine Formel erzeugt (Zusicherung A3). Aus diesen Formeln und einem ausgezeichneten Prädikat wird ein Prädikat mit Parameter erzeugt (Zusicherung A4) und schließlich induktiv geprüft, ob dieses Prädikat ein allgemeines Prädikat ist (Zusicherung A5). Falls ja, wird aus diesem *wp* berechnet und zurückgegeben.
Durch Betrachtung der Stellen, an denen `no_success` zurückgegeben wird, können die Gründe dafür erkannt werden, daß der Algorithmus *wp* nicht berechnen kann. Diese sind:

1. es werden keine 2 aufeinanderfolgenden strukturgleichen Prädikate gefunden

oder 2. die Sequenz der *n* Prädikate, beginnend mit den ersten 2 strukturgleichen, ist nicht strukturgleich

oder 3. die Sequenz der *n* Prädikate, beginnend mit den ersten 2 strukturgleichen, ist zwar strukturgleich, doch können aus den extrahierten Zahlenfolgen keine darauf passenden Formeln generiert werden

oder 4. die Sequenz der *n* Prädikate, beginnend mit den ersten 2 strukturgleichen, ist strukturgleich, und aus den extrahierten Zahlenfolgen können darauf passende Formeln generiert werden, doch das daraus generierte Prädikat ist kein allgemeines Prädikat

```
-- Input:   while-loop while B loop S end loop;
--         postcondition N
-- Output:  wp(while-loop, N) or no_success;
function wp_for_while(B: boolean_expression; S: statement;
                      N: predicate)
return result_type is
  max_tries:   constant := 4; -- Ende Abschnitt 4.2.5, Frage 1
  n:           constant := 5; -- Ende Abschnitt 4.2.5, Frage 2
  P0, P1, ... : predicate;
  P(i):        predicate_with_parameter_i;
  m:           natural := 0;
  var_num      natural;
  s:           array(1 .. var_num, 1 .. n-1) of integer;
  f:           array(1 .. var_num) of formula;
begin
  P0 := false;
  for i in 1..max_tries loop
        Pi := ¬B ∧ N ∨ B ∧ wp(S, Pi-1);
        DNF(Pi);
        make_si(Pi);
        if S(Pi, Pi-1)
        then  m := i;
              exit;
        end if;
  end loop;
  if m=0 then return no_success; end if;
  -- A1: min{k | S(Pk, Pk+1)} = m - 1
  for i in m+1 .. m+n-2 loop
        Pi := ¬B ∧ N ∨ B ∧ wp(S, Pi-1);
        DNF(Pi);
        make_si(Pi);
  end loop;
  -- Pm-1, ..., Pm+n-2 ist eine Sequenz von n Prädikaten
  if not S(Pm-1, ..., Pm+n-2) then return no_success; end if;
  -- Pm-1, ..., Pm+n-2 ist eine Sequenz von n strukturgleichen
  -- Prädikaten
  var_num := |Var(t3(Pm-1))|;
  for j in m .. m+n-2 loop
        for i in 1 .. var_num loop
              s(i, j-m+1) := hi(mgu(t3(Pm-1), t1(Pj)));
        end loop;
  end loop;
  -- A2: var_num Zahlenfolgen s(i) der Länge n-1
  for i in 1 .. var_num loop
        f(i) := generate_formula(s(i));
        if f(i) = no_success then return no_success; end if;
  end loop;
  -- A3: var_num Formeln f(i) zu den Zahlenfolgen s(i)
  P(i) := Definition 4.17;
  -- A4: P(i) might be a general predicate;
  --     this must be checked by induction
  if not (P(0) = P0 and P(i+1) = ¬B ∧ N ∨ B ∧ wp(S, P(i)))
  then return no_success;
  end if;
  -- A5: P(i) is a general predicate
  return (∃i≥0:P(i));
end wp_for_while;
```

Verwendete Typen, Funktionen und Prozeduren

`boolean_expression`	ein geeigneter Typ für Boolesche Ausdrücke
`statement`	ein geeigneter Typ für Anweisungen
`predicate`	ein geeigneter Typ für Prädikate
`predicate_with_parameter_i`	ein geeigneter Typ für Prädikate mit dem Parameter *i*
`result_type`	ein Typ bestehend aus Prädikaten und dem Literal `no_success`
`formula`	ein Typ für Formeln, wie sie in Abschnitt 4.2.6 generiert werden
function `Var(p: predicate)` **return** `set_of_var`	eine Funktion nach Definition 2.2
function `t3(p: predicate)` **return** `predicate`	eine Funktion nach Definition 4.12
function `t1(p: predicate)` **return** `predicate`	eine Funktion nach Definition 4.6
procedure `DNF(p: in out predicate);`	eine Prozedur zur Transformation eines Prädikates *p* in DNF
procedure `make_si(P: in out predicate);`	eine Prozedur zur Transformation eines Prädikates *p* nach Abschnitt 4.2.7
function `S(P: sequence_of_predicate)` **return** `boolean`	eine Funktion zur Feststellung der Strukturgleichheit nach Definition 4.6
function `mgu(P: sequence_of_predicate)` **return** `mgu_type`	eine Funktion zur Unifikation nach Definition 4.9
function `hi(u: mgu_type)` **return** `integer`	eine Funktion nach Definition 4.15
Function `generate_formula(s:sequence_of_integer)` **return** `formula`	eine Funktion, die zu einer Sequenz von Zahlen eine Formal generiert nach Abschnitt 4.2.6

4.2.5 Transformation der Terme und Reduktion auf Unifikation

Der Zusammenhang zwischen $S(P)$ und $U(P)$ ist nicht so einfach. Genauer gesagt besteht zwischen $S(P)$ und $U(P)$ keine der 12 einfachen aus folgendem regulären Ausdruck ableitbaren Beziehungen: ['¬'] '$U(P)$' ('⇒' | '⇐' | '≡') ['¬'] '$S(P)$'. Dazu reicht es zu zeigen, daß keine der folgenden 4 Beziehungen gilt:

$U(P) \Rightarrow S(P)$; Gegenbeispiel: $P = (x > 1, y > 1)$
$U(P) \Rightarrow \neg S(P)$; Gegenbeispiel: $P = (x > 1, x > 1)$
$\neg U(P) \Rightarrow S(P)$; Gegenbeispiel: $P = (x > 1, y > 2)$
$S(P) \Rightarrow U(P)$; Gegenbeispiel: $P = (x > 1, x > 2)$

Die Struktur eines Prädikates wird im Wesentlichen durch die Folge und die Beziehungen zwischen den Funktions-, Relations- und Variablensymbolen bestimmt, nicht von den Konstanten. Hier muß man zwischen Konstanten als nullstelligen Funktionssymbolen und mindestens einstelligen Funktionssymbolen unterscheiden. Um die Unifizierbarkeit darauf anzuwenden, muß man also die Funktions-, Relations- und Variablensymbole festhalten bzw. als konstant betrachten und die Konstanten als variabel. Die Funktions- und Relationssymbole sind konstant. Um die Variablen konstant zu machen, wird jedes Variablensymbol '*v*' durch

das Funktionssymbol $'v'(a)$ ersetzt, wobei *a* ein neues Variablensymbol ist. Damit wird verhindert, daß z.B. $P = (x > 1, y > 1)$ strukturgleich sind. Dann nach dieser Transformation sieht *P* so aus: $P = (x(a) > 1, y(a) > 1)$.
Um die Konstanten variabel zu machen, wird jede Zahl in jedem Prädikat der Sequenz bis auf eines durch ein neues Variablensymbol $'h'ij$ ersetzt. Dabei ist *i* der Index des aktuellen Prädikats und *j* der Index der aktuellen Zahl in P_i. Dadurch wird sichergestellt, daß auch gleiche Zahlen durch verschiedene Variablensymbole ersetzt werden. In einem ausgezeichneten Prädikat P_s der Sequenz wird nicht diese, sondern eine andere Transformation durchgeführt. In P_s wird die *n.* Zahl durch *n* ersetzt, so daß im transformierten Prädikat alle Zahlen paarweise verschieden sind. Aus $P = (x > 2, x > 4)$ wird durch diese beiden Transformationen (o.B.d.A. ist $s = 0$) $P = (x(a) > 1, x(a) > h_{21})$. Jetzt wird die Strukturgleichheit durch die Unifizierbarkeit der transformierten Sequenz definiert.

Definition 4.6 (Strukturgleichheit): Sei *P* eine Sequenz von Prädikaten.
P heißt *strukturgleich* ($S(P)$), genau dann wenn $U(t(P))$ gilt, also $S(P) :\equiv U(t(P))$.
Dabei ist $t(P) := (t(P_0), t(P_1), \ldots, t(P_n))$, $t(P_i) :\equiv t_2(t_1(P_i)), i = 0 \ldots n$,

$t_1(P_i) :\equiv (P_i)^v_{v(a)}$ für jedes Variablensymbol *v*, wobei *a* eine neue Variable ist, d.h.

$$a \notin \bigcup_{i=0}^{n} free(P_i).$$

$pref(P, n)$ ist das Präfix des Prädikats *P* als String bis zum Zahlenliteral *n*. Die Funktion *o*: $PS \times \mathbb{Z} \mapsto \mathbb{N}$ mit $o(P, n) := 1 + (N : z \in \mathbf{Z} : z \in pref(P, n))$ ordnet jedem Zahlenliteral *n* in einem Prädikat *P* die Zahl $o(n)$ zu, so daß *n* das $o(n)$. Vorkommen eines Zahlenliterals in *P* ist. Im Folgenden wird das 1. Argument der Funktion *o* weggelassen, wenn es aus dem Zusammenhang hervorgeht.

$$t_2(P_i) :\equiv \begin{cases} (P_i)^c_{o(c)}, & falls\ i = s \\ (P_i)^c_{hi,o(c)}, & falls\ i \neq s \end{cases}$$ für jedes Zahlenliteral *c*, $s \in \{0 \ldots n\}, i = 0 \ldots n$.

•

Beispiel:

$$P_0 \equiv x > 1 \wedge y > 1,\ P_1 \equiv x > 2 \wedge y > 2,\ P_2 \equiv x > 3 \wedge y > 4.$$

O.B.d.A. kann man $s = 0$ nehmen.
$t(P_0) \equiv x(a) > 1 \wedge y(a) > 2$, $t(P_1) \equiv x(a) > h_{11} \wedge y(a) > h_{12}$, $t(P_2) \equiv x(a) > h_{21} \wedge y(a) > h_{22}$
Es gilt $U(t(P))$ mit dem Unifikator $\{h_{11} \mapsto 1, h_{12} \mapsto 2, h_{21} \mapsto 1, h_{22} \mapsto 2\}$ und damit $S(P)$.

•

Lemma 4.7: *S* ist eine Äquivalenzrelation. Falls *P* die leere Sequenz ε ist oder nur 1 Prädikat enthält, gilt $S(P)$.
Beweis: *S* wird durch die Unifizierbarkeit definiert, die wiederum durch die Gleichheit definiert ist, was eine Äquivalenzrelation ist.

$$S(\varepsilon) \equiv (\forall P_1, P_2 \in \varepsilon : S(P_1, P_2)) \equiv true$$
$$S((P)) \equiv (\forall P_1, P_2 \in (P) : S(P_1, P_2)) \equiv S(P, P) \equiv true$$

•

Lemma 4.8: Die Folge der transformierten Prädikate enthält keine Variable aus der Folge der ursprünglichen Prädikate, und zusätzlich enthält die Transformation des ausgezeichneten Prädikates keine der Variablen $h_{i,j}$, d. h.

$$(\forall v \in \bigcup_{i=0}^{n} free(P_i) : v \notin \bigcup_{i=0}^{n} free(t(P_i))) \wedge (\forall i, j : h_{i,j} \notin t(P_s)).$$

Beweis: Die Behauptung folgt direkt aus Definition 4.6.

•

Damit ist die Strukturgleichheit definiert. Jetzt geht es darum, die Unterschiede in den einzelnen Prädikaten herauszufinden, falls $S(P)$ gilt. Dazu wird ein allgemeinster Unifikator *mgu* (most general unifier) von $t(P)$ berechnet. Dieser existiert, da diese Berechnung nur durchgeführt wird, falls $S(P)$, d.h. $U(t(P))$ gilt. *mgu* ist eine Funktion von einer endlichen, nicht leeren Folge von Prädikaten, deren Anwendung eine endliche Funktion von der Menge der Variablen in $t(P)$ in die Menge der Terme *T* liefert, d.h.

Definition 4.9: $mgu : \bigtimes_{i=0}^{n} PS \rightarrow \left(\bigcup_{i=0}^{n} free(t(P_i)) \quad \rightarrow T \right)$, $\sigma := mgu(t(P_0), \ldots, t(P_n))$,

$$\sigma = \{h_{i,j} \mapsto u_j\} \cup \{a \mapsto u_a\}, \text{ wobei } h_{i,j}, a \in \bigcup_{i=0}^{n} free(t(P_i)) \text{ und } u_x \in T.$$

•

Lemma 4.10: $\{a \mapsto u_a\} \in \sigma \Rightarrow u_a = a$.

Beweis durch Induktion über den induktiven Aufbau von P_s:

Induktionsanfang: P_s sei Konstante;

$P_s = c_1, c_1 \in \mathbf{Z}$
$\Rightarrow$ Definition Strukturgleichheit
$t(P_s) = 1$ (o.B.d.A)
$\Rightarrow$ Definition Unifizierbarkeit, Definition Strukturgleichheit
$t(P_i) = h_{i,1}, i \neq s$
$\Rightarrow$ Definition Strukturgleichheit
$P_i = c_2, c_2 \in \mathbf{Z}, i \neq s$
$\Rightarrow$ Voraussetzung
$a \notin free(t(P_s)) \cup free(t(P_i))$
$\Rightarrow$ Unifikation
$\{a \mapsto u_a\} \notin \sigma$
$\Rightarrow$ Logik
$\{a \mapsto u_a\} \in \sigma \Rightarrow u_a = a$

P_s sei Variable;

$P_s = x$
$\Rightarrow$ Definition Strukturgleichheit
$t(P_s) = x(a)$
$\Rightarrow$ Definition Unifizierbarkeit, Definition Strukturgleichheit
$t(P_i) = h_{i,1} \vee t(P_i) = x(u), i \neq s, u \in T$
$\Rightarrow$ Definition Strukturgleichheit
$P_i = c \vee P_i = x, i \neq s, c \in \mathbf{Z}$
$\Rightarrow$ Definition Strukturgleichheit
$t(P_i) = h_{i,1} \vee t(P_i) = x(a), i \neq s$
$\Rightarrow$ Voraussetzung

$a \in free(t(P_s)) \cup free(t(P_i))$
$\Rightarrow$ Unifikation
$\{a \mapsto u_a\} \in \sigma \wedge u_a = a$

Induktionsschritt: P_s sei $f(u_1,\ldots,u_m), u_1,\ldots,u_m \in T$, P_i sei $f(s_1,\ldots,s_m), s_1,\ldots,s_m \in T, i \neq s, i = 0\ldots n$ und f sei ein m-stelliges Funktionssymbol;
Induktionsvoraussetzung $\{a \mapsto u_a\} \in mgu(t(u_1),t(s_1)) \cup \cdots \cup mgu(t(u_m),t(s_m)) \Rightarrow u_a = a$.
Zu zeigen

$\{a \mapsto u_a\} \in mgu(t(f(u_1,\ldots,u_m)),t(f(s_1,\ldots,s_m))) \Rightarrow u_a = a$
$\equiv$ Definition Strukturgleichheit
$\{a \mapsto u_a\} \in mgu(f(u_1,\ldots,u_m)^{v}_{v(a)}, f(s_1,\ldots,s_m)^{v}_{v(a)}) \Rightarrow u_a = a$
$\equiv$ Lemma 2.9
$\{a \mapsto u_a\} \in mgu(f(u_{1v(a)}^{v},\ldots,u_{mv(a)}^{v}), f(s_{1v(a)}^{v},\ldots,s_{mv(a)}^{v})) \Rightarrow u_a = a$
$\equiv$ Definition Strukturgleichheit
$\{a \mapsto u_a\} \in mgu(f(t(u_1),\ldots,t(u_m)), f(t(s_1),\ldots,t(s_m))) \Rightarrow u_a = a$
$\Leftarrow$ Definition Unifikator
$\{a \mapsto u_a\} \in mgu(t(u_1),t(s_1)) \cup \quad \cup mgu(t(u_m),t(s_m)) \Rightarrow u_a = a$
$\equiv$ Induktionsvoraussetzung
true

•

Lemma 4.11: Die Anwendung von σ auf alle transformierten Prädikate der Sequenz ergibt $t(P_s)$, d.h. $(\forall i : 0 \le i \le n : \sigma(t(P_i)) \equiv t(P_s))$.
Beweis:

$(\forall i, j : 0 \le i, j \le n : \sigma(t(P_i)) \equiv \sigma(t(P_j)))$ nach Definition von σ
$\Rightarrow$ mit $j = s$
$(\forall i : 0 \le i \le n : \sigma(t(P_i)) \equiv \sigma(t(P_s)))$
$\equiv$ Definition σ
$(\forall i : 0 \le i \le n : \sigma(t(P_i)) \equiv t(P_s)^{h_{i,j},a}_{u_{i,j},u_a})$
$\equiv$ Lemma 4.8
$(\forall i : 0 \le i \le n : \sigma(t(P_i)) \equiv t(P_s)^{a}_{u_a})$
$\equiv$ Lemma 4.10
$(\forall i : 0 \le i \le n : \sigma(t(P_i)) \equiv t(P_s))$

•

Definition 4.12: $t_3(P_s) :\equiv (t(P_s))^{o(c)}_{ho(c)}$, d.h. in $t(P_s)$ wird jedes Zahlenliteral $o(c)$ durch das neue Variablensymbol 'h' $o(c)$ ersetzt.

•

Das Prädikat $t_3(P_s)$ wird mit jedem Prädikat aus der Sequenz $t_1(P_i), i = 0\ldots n$ unifiziert. Das ergibt eine Sequenz *MGU* von Unifikatoren, aus denen die Zahlenfolgen extrahiert werden können, in denen sich die einzelnen Prädikate der Sequenz *P* unterscheiden.

Definition 4.13: $MGU :\equiv (mgu(t_3(P_s),t_1(P_i)), i = 0\ldots n, i \neq s$.

•

Lemma 4.14: Jedes Element der Sequenz *MGU* ist eine Menge der Form

$$\{h_{o(c)} \mapsto m_j \mid h_{o(c)} \in \mathit{free}(t_3(P_s)), m_j \in \mathbf{Z}\}.$$

Beweis: sei $\rho_i := mgu(t_3(P_s), t_1(P_i)), i = 0 \ldots n$. Nach Definition von *mgu* gilt dann $\rho_i(t_3(P_s)) \equiv \rho_i(t_1(P_i))$, und zu zeigen ist $\rho_i = \{h_{o(c)} \mapsto m_j \mid h_{o(c)} \in Var(t_3(P_s)), m_j \in \mathbf{Z}\}$, d.h.

$(t_3(P_s))_m^{h_{o(c)}} \equiv (t_1(P_i))_m^{h_{o(c)}}$

≡ Definition 4.12, $h_{o(c)} \notin t_1(P_i)$

$((t(P_s))_{h_{o(c)}}^{o(c)})_m^{h_{o(c)}} \equiv t_1(P_i)$

≡ Substitutionslemma

$(t(P_s))_m^{o(c)} \equiv t_1(P_i)$

≡ Leibniz, Strukturgleichheit

$\sigma(t_2((t(P_s))_m^{o(c)})) \equiv \sigma(t(P_i))$

≡ Lemma 4.11

$\sigma(t_2((t(P_s))_m^{o(c)})) \equiv t(P_s)$

≡ Definition Strukturgleichheit

$\sigma(((t(P_s))_m^{o(c)})_{o(m)}^m) \equiv t(P_s)$

≡ Substitutionslemma

$\sigma((t(P_s))_{o(m)}^{o(c)}) \equiv t(P_s)$

≡ da durch die Substitution $o(m)$ an der gleichen Stelle wie $o(c)$ steht, gilt $o(c)=o(m)$

$\sigma(t(P_s)) \equiv t(P_s)$

≡ Lemma 4.11

true

•

Die Zahlenfolgen können aus MGU extrahiert werden.

Definition 4.15:

$$h_k(\{h_{o(c)} \mapsto m_j \mid h_{o(c)} \in Var(t_3(P_s))\}) := \begin{cases} m_j, & \mathit{falls}\ h_k \mapsto m_j \in \{h_{o(c)} \mapsto m_j \mid h_{o(c)} \in Var(t_3(P_s))\} \\ & \mathit{undefiniert\ sonst} \end{cases}$$

, d.h. diese Funktion extrahiert aus einem *mgu* diejenige Zahl, auf die die Variable h_k durch den *mgu* abgebildet wird.

•

Beispiel: $h_2(\{h_1 \mapsto 2, h_2 \mapsto 4, h_3 \mapsto 6\}) = 4$.

•

Zu jeder Zahl $o(c)$ in $t(P_s)$ wird eine Zahlenfolge $s(o(c), i), i = 0 \ldots n$ definiert durch

Definition 4.16: $s(o(c), i) := h_{o(c)}(MGU_i)$ für *i*=0...*n*.

•

Jetzt wird aus der Sequenz $s(o(c), i)$ eine Formel $s'(o(c), i)$ generiert, die an den Stellen 0...*n* mit $s(o(c), i)$ übereinstimmt. Für jede endliche Zahlenfolge gibt es die triviale Formel, die diese Bedingung erfüllt, nämlich die durch Fallunterscheidung. Außerdem gibt es noch unendlich viele nicht triviale Formeln, die die $n+1$ Punkte als Stützstellen eines ganzzahligen Polynoms betrachten. Doch diese beiden Ansätze liefern im Allgemeinen kein allgemeines Prädikat. Hier wird das allgemeine Prädikat durch folgende Annahmen erzeugt:

Es wird angenommen, daß es sich um eine arithmetische Reihe handelt, wenn die Differenzen 1. Grades konstant sind. Es wird angenommen, daß es sich um ein ganzzahliges quadratisches Polynom handelt, wenn die Differenzen 2. Grades konstant sind. Es wird angenommen, daß es sich um eine geometrische Reihe handelt, wenn die Quotienten konstant sind. Man kann beliebige weitere Gesetzmäßigkeiten dieser Art einbauen, aber damit kommt man für die meisten Fälle aus, wenn es sich nicht um Schleifen in ausgesprochen mathematischen Algorithmen handelt. Das allgemeine Prädikat $P(i)$ ergibt sich aus Substitution der Zahlen $o(c)$ in $t(P_s)$ durch die Formel $s'(o(c),i)$. Die Konstruktion von *s'* ist Gegenstand des nächsten Abschnitts.

Definition 4.17: Das allgemeine Prädikat für alle Zahlen $o(c)$ in $t(P_s)$ ist

$$P(i) :\equiv t_1^{-1}((t(P_s))^{o(c)}_{s'(o(c),i)}),$$

wobei t_1^{-1} die Umkehrtransformation von t_1 ist. Diese existiert und ist eindeutig, was direkt aus der Definition der Strukturgleichheit folgt.

•

Lemma 4.18: Mit ρ_i aus Definition 4.13 und Lemma 4.14 gilt

$$\rho_i(t_3(P_s)) \equiv (t(P_s))^{o(c)}_{s(o(c),i)}.$$

Beweis:

$\rho_i(t_3(P_s))$
≡ Definition 4.13, Lemma 4.14
$(t_3(P_s))^{h_{o(c)}}_m$
≡ Definition 4.12
$((t(P_s))^{oc}_{h_{o(c)}})^{h_{o(c)}}_m$
≡ Substitutionslemma
$(t(P_s))^{o(c)}_m$
≡ Definition 4.15, Definition 4.16
$(t(P_s))^{o(c)}_{s(o(c),i)}$

•

Satz 4.19: $(\forall i : 0 \le i \le n : P(i) \equiv P_i)$.

Beweis:

$P(i) \equiv P_i$
≡ Definition 4.17
$t_1^{-1}((t(P_s))^{o(c)}_{s'(o(c),i)}) \equiv P_i$
≡ Konstruktion *s'*
$t_1^{-1}((t(P_s))^{o(c)}_{s(o(c),i)}) \equiv P_i$
≡ Definition Strukturgleichheit
$(t_2(P_s))^{o(c)}_{s(o(c),i)}) \equiv P_i$
≡ Leibniz, Definition Strukturgleichheit
$(t(P_s))^{o(c)}_{s(o(c),i)}) \equiv t_1(P_i)$
≡ Lemma 4.18
$\rho_i(t_3(P_s)) \equiv t_1(P_i)$
≡ Definition Strukturgleichheit, Lemma 4.14, $h_{o(c)} \notin free(t_1(P_i))$

$\rho_i(t_3(P_s)) \equiv \rho_i(t_1(P_i))$
$\equiv$ Lemma 4.14
true

•

Im Allgemeinen ist es so, daß die betrachtete explizite Anfangssequenz nicht mit P_0 beginnt, sondern mit einem späteren Prädikat. Dazu ein kleines Beispiel für $x \in \mathbb{N}$:

$$P_0 \equiv false,\ P_1 \equiv x > 0,\ P_2 \equiv 2x > 0,\ P_3 \equiv 3x > 0$$

Diese Sequenz ist nach der Definition nicht strukturgleich, da die einzelnen Elemente nach der Transformation *t* nicht unifizierbar sind. Wenn man jedoch die Sequenz mit P_2 beginnen läßt, erhält man als allgemeines Prädikat $P(i) \equiv ix > 0$. Nach Konstruktion paßt dieses Prädikat auf die betrachtete Sequenz, die hier mit P_2 beginnt. Man muß also noch prüfen, ob $P(i)$ auch auf die nicht betrachteten Prädikate, hier P_0 und P_1, paßt, was hier der Fall ist, wie man leicht nachprüfen kann. Das 1. Prädikat der zu betrachteten Sequenz ist das 1. Prädikat, das strukturgleich zu seinem Nachfolger ist. Es hat den Index $\min\{i | S(P_i, P_{i+1})\}$. Es kann natürlich vorkommen, daß keine Sequenz strukturgleicher Prädikate entsteht. Damit ergeben sich 2 Fragen:

1. Wie lange soll man suchen, bis $\min\{i | S(P_i, P_{i+1})\}$ gefunden wird? Für die in den Beispielen betrachteten und einige weitere Schleifen gilt $i \le 4$.
2. Wie viele Prädikate soll man betrachten? Das kommt auf die Komplexität der zu erzeugenden Formel *s'* an. Wenn man eine Formel für konstante Differenzen / Quotienten *n*. Grades erzeugen will, muß man *n*+2 Prädikate betrachten.

4.2.6 Die Generierung der Formeln aus der gegebenen Sequenz

Gegeben ist eine Sequenz $s = (s_0, \ldots, s_n)$, gesucht ist eine Funktion $s'(i)$, so daß $(\forall i : 0 \le i \le n : s'(i) = s_i)$ gilt (hier wird das 1. Argument von *s* aus Definition 4.16 weggelassen, da es in diesem Zusammenhang keine Rolle spielt, und das 2. Argument wird als Index geschrieben). Unter den beliebig vielen Funktionen werden nur ganzzahlige Polynome bis zum Grad 2, geometrische Reihen und Kombinationen daraus betrachtet, die im Folgenden definiert werden. Das ist eine Heuristik, da sich an mehreren Beispielen gezeigt hat, daß die erzeugten, endlichen Zahlenfolgen die Eigenschaften der folgenden Definition haben.

Definition 4.20:

1. $D^0(s) :\equiv (\forall i : 0 \le i \le n-1 : s_i = s_{i+1}) \wedge n \ge 1$, d.h. *s* ist eine Konstante Sequenz
2. $D(s) :\equiv (\forall i : 0 \le i \le n-2 : 2s_{i+1} = s_i + s_{i+2}) \wedge n \ge 2$, d.h. die Sequenz der Differenzen von *s* ist konstant. Dann ist *s* eine arithmetische Reihe.
3. $D^2(s) :\equiv (\forall i : 0 \le i \le n-3 : 3(s_{i+2} - s_{i+1}) = s_{i+3} - s_i) \wedge n \ge 3$, d.h. die Sequenz der Differenzen der Differenzen von *s* ist konstant
4. $Q^0(s) :\equiv D^0(s)$, d.h. *s* ist eine Konstante Sequenz
5. $Q(s) :\equiv (\forall i : 0 \le i \le n-1 : s_{i+1}^2 = s_i \cdot s_{i+2}) \wedge n \ge 2$, d.h. die Sequenz der Quotienten von *s* ist konstant. Dann ist *s* eine geometrische Reihe.
6. $Q^2(s) :\equiv (\forall i : 0 \le i \le n-3 : \left(\frac{s_{i+2}}{s_{i+1}}\right)^3 = \frac{s_{i+3}}{s_i}) \wedge n \ge 3$, d.h. die Sequenz der Quotienten der Quotienten von *s* ist konstant

7. $DQ(s) :\equiv (\forall i : 0 \le i \le n-3 : (s_{i+2} - s_{i+1})^2 = (s_{i+1} - s_i)(s_{i+3} - s_{i+2})) \wedge n \ge 3$, d.h. die Sequenz der Quotienten der Differenzen von *s* ist konstant
8. $QD(s) :\equiv (\forall i : 0 \le i \le n-3 : 2\frac{s_{i+2}}{s_{i+1}} = \frac{s_{i+3}}{s_{i+2}} + \frac{s_{i+1}}{s_i}) \wedge n \ge 3$, d.h. die Sequenz der Differenzen der Quotienten von *s* ist konstant

•

Beispiele:

1. (2,2,2) hat die Eigenschaft D^0.
2. (3,6,9,12) hat die Eigenschaft *D*.
3. (2,4,7,11,16) hat die Eigenschaft D^2.
4. (1,1,1) hat die Eigenschaft Q^0.
5. (1,3,9,27,819) hat die Eigenschaft *Q*.
6. (2,4,16,128,2048) hat die Eigenschaft Q^2.
7. (1,3,7,15,31) hat die Eigenschaft DQ.
8. (2,4,16,96,768) hat die Eigenschaft *QD*.

Die in den Definitionen auftretenden Konjunkte $n \ge a, a \in \{1,2,3\}$, sorgen dafür, daß nur genügend lange Sequenzen mit den entsprechenden Eigenschaften versehen werden. Denn es ist z.B. sinnlos zu sagen, die Sequenz (2,4) hat die Eigenschaft *D*, was rein formal ohne $n \ge 2$ gelten würde, da dann in der Definition von *D* über den leeren Bereich allquantifiziert würde.

•

Die in Definition 4.20 angegebenen Prädikate über einer Sequenz *s* sind relativ einfach zu entscheiden. Dazu prüft man, ob *s* die Eigenschaft *E* hat, wobei *E* die in Definition 4.20 angegebenen Prädikate (in der Reihenfolge) durchläuft, bis *s* eine der Eigenschaften hat oder alle Prädikate geprüft sind. Im letzteren Fall endet das Verfahren erfolglos. Falls *s* eine der Eigenschaften hat, kann mit Hilfe des folgenden Lemmas die Formel $s'(i)$ erzeugt werden.

Lemma 4.21:

1. $D^0(s) \Rightarrow s'(i) = s_0$
2. $D(s) \Rightarrow s'(i) = (s_1 - s_0)i + s_0$
3. $D^2(s) \Rightarrow s'(i) = \frac{s_0 - 2s_1 + s_2}{2} i^2 + \frac{-3s_0 + 4s_1 - s_2}{2} i + s_0$
4. $Q^0(s) \Rightarrow s'(i) = s_0$
5. $Q(s) \Rightarrow s'(i) = \left(\frac{s_1}{s_0}\right)^i \cdot s_0$
6. $Q^2(s) \Rightarrow s'(i) = \sqrt{\frac{s_0 \cdot s_2}{s_1^2}}^{i^2} \cdot \sqrt{\frac{s_1^4}{s_0^3 \cdot s_2}}^i \cdot s_0$
7. $DQ(s) \Rightarrow s'(i) = \left(\frac{s_2 - s_1}{s_1 - s_0}\right)^i (s_1 - s_0) - s_0$
8. $QD(s) \Rightarrow s'(i) = s_0 \prod_{j=0}^{i-1} \left(\left(\frac{s_2}{s_1} - \frac{s_1}{s_0}\right) j + \frac{s_1}{s_0}\right)$

Beweis durch Einsetzen der rechten Seiten von $s'(i)$ in die Formeln von Definition 4.20:

1. $s_0 = s_0$

2. $2((s_1 - s_0)(i+1) + s_0) = (s_1 - s_0)i + s_0 + (s_1 - s_0)(i+2) + s_0$

3.

$$3(\frac{s_0 - 2s_1 + s_2}{2}(i+2)^2 + \frac{-3s_0 + 4s_1 - s_2}{2}(i+2) + s_0 -$$
$$(\frac{s_0 - 2s_1 + s_2}{2}(i+1)^2 + \frac{-3s_0 + 4s_1 - s_2}{2}(i+1) + s_0)) =$$
$$\frac{s_0 - 2s_1 + s_2}{2}(i+3)^2 + \frac{-3s_0 + 4s_1 - s_2}{2}(i+3) + s_0 -$$
$$(\frac{s_0 - 2s_1 + s_2}{2}i^2 + \frac{-3s_0 + 4s_1 - s_2}{2}i + s_0)$$

4. $s_0 = s_0$

5. $$\left(\frac{s_1}{s_0}\right)^{2(i+1)} \cdot s_0 = \left(\frac{s_1}{s_0}\right)^{i} \cdot s_0 \cdot \left(\frac{s_1}{s_0}\right)^{(i+2)} \cdot s_0$$

6. $$\left(\frac{\sqrt{\frac{s_0 \cdot s_2}{s_1^2}}^{(i+2)^2} \cdot \sqrt{\frac{s_1^4}{s_0^3 \cdot s_2}}^{(i+2)} \cdot s_0}{\sqrt{\frac{s_0 \cdot s_2}{s_1^2}}^{(i+1)^2} \cdot \sqrt{\frac{s_1^4}{s_0^3 \cdot s_2}}^{(i+1)} \cdot s_0}\right)^3 = \frac{\sqrt{\frac{s_0 \cdot s_2}{s_1^2}}^{(i+3)^2} \cdot \sqrt{\frac{s_1^4}{s_0^3 \cdot s_2}}^{(i+3)} \cdot s_0}{\sqrt{\frac{s_0 \cdot s_2}{s_1^2}}^{i^2} \cdot \sqrt{\frac{s_1^4}{s_0^3 \cdot s_2}}^{i} \cdot s_0}$$

7.

$$\left(\left(\frac{s_2 - s_1}{s_1 - s_0}\right)^{i+2}(s_1 - s_0) - s_0 - \left(\left(\frac{s_2 - s_1}{s_1 - s_0}\right)^{i+1}(s_1 - s_0) - s_0\right)\right)^2 =$$
$$\left(\left(\frac{s_2 - s_1}{s_1 - s_0}\right)^{i+1}(s_1 - s_0) - s_0 - \left(\left(\frac{s_2 - s_1}{s_1 - s_0}\right)^{i}(s_1 - s_0) - s_0\right)\right) \cdot$$
$$\left(\left(\frac{s_2 - s_1}{s_1 - s_0}\right)^{i+3}(s_1 - s_0) - s_0 - \left(\left(\frac{s_2 - s_1}{s_1 - s_0}\right)^{i+2}(s_1 - s_0) - s_0\right)\right)$$

8. $$2\frac{s_0\prod_{j=0}^{i+1}\left(\left(\frac{s_2}{s_1} - \frac{s_1}{s_0}\right)j + \frac{s_1}{s_0}\right)}{s_0\prod_{j=0}^{i}\left(\left(\frac{s_2}{s_1} - \frac{s_1}{s_0}\right)j + \frac{s_1}{s_0}\right)} = \frac{s_0\prod_{j=0}^{i+2}\left(\left(\frac{s_2}{s_1} - \frac{s_1}{s_0}\right)j + \frac{s_1}{s_0}\right)}{s_0\prod_{j=0}^{i+1}\left(\left(\frac{s_2}{s_1} - \frac{s_1}{s_0}\right)j + \frac{s_1}{s_0}\right)} + \frac{s_0\prod_{j=0}^{i+1}\left(\left(\frac{s_2}{s_1} - \frac{s_1}{s_0}\right)j + \frac{s_1}{s_0}\right)}{s_0\prod_{j=0}^{i-1}\left(\left(\frac{s_2}{s_1} - \frac{s_1}{s_0}\right)j + \frac{s_1}{s_0}\right)}$$

Um die Gleichheiten zu zeigen, sind umfangreiche algebraische symbolische Operationen erforderlich. Dieses von Hand durchzuführen ist mühsam und sehr fehleranfällig. Daher liegt es nahe zu versuchen, die Gleichungen per Computer umzuformen und dann die syntaktische Gleichheit der entstandenen Terme zu prüfen. Das kann tatsächlich gemacht werden. Die Gleichheiten 1. – 7. können von Mathematica vollständig gezeigt werden (Gleichheit 8 nur teilweise, der Rest von Hand). Daher ist es nicht nötig, die Umformungen explizit aufzuführen, und zwar nicht aus Platzgründen oder weil es keine neuen Einsichten oder Erkenntnisse

bezüglich Beweistechniken bringt (was hier auch zutrifft), sondern einfach, weil die Gleichheit rein mechanisch gezeigt werden kann. Das ist analog zu einer arithmetischen Gleichheit, die leicht mit einem Taschenrechner geprüft werden kann und die auch jeder glaubt, ohne von Hand nachzurechnen.

•

Beispiel:

$$P_0 \equiv y < -1 \wedge b(y) < b(x) \wedge x + 1 = y$$
$$P_1 \equiv b(y) < b(2x) \wedge y < 0 \wedge x - 3 = y$$
$$P_2 \equiv b(y) < b(4x) \wedge x - 7 = y \wedge y < 3$$
$$P_3 \equiv x - 11 = y \wedge b(y) < b(8x) \wedge y < 8$$
$$P_4 \equiv x - 15 = y \wedge y < 15 \wedge b(y) < b(16x)$$

$\min\{i | S(P_i, P_{i+1})\} = 1$, also beginnt die betrachtete Sequenz bei P_1.

$$t_1(P_1) \equiv b(y(a)) < b(2x(a)) \wedge y(a) < 0 \wedge x(a) - 3 = y(a)$$
$$t_1(P_2) \equiv b(y(a)) < b(4x(a)) \wedge x(a) - 7 = y(a) \wedge y(a) < 3$$
$$t_1(P_3) \equiv x(a) - 11 = y(a) \wedge b(y(a)) < b(8x(a)) \wedge y(a) < 8$$
$$t_1(P_4) \equiv x(a) - 15 = y(a) \wedge y(a) < 15 \wedge b(y(a)) < b(16x(a))$$

$$t(P_1) \equiv b(y(a)) < b(1x(a)) \wedge y(a) < 2 \wedge x(a) - 3 = y(a)$$
$$t(P_2) \equiv b(y(a)) < b(h_{21}x(a)) \wedge x(a) - h_{22} = y(a) \wedge y(a) < h_{23}$$
$$t(P_3) \equiv x(a) - h_{31} = y(a) \wedge b(y(a)) < b(h_{32}x(a)) \wedge y(a) < h_{33}$$
$$t(P_4) \equiv x(a) - h_{41} = y(a) \wedge y(a) < h_{42} \wedge b(y(a)) < b(h_{43}x(a))$$

Es gilt $U(t(P_1,\ldots,P_4))$, also $S(P_1,\ldots,P_4)$ mit

$$mgu = \{h_{21} \mapsto 1, h_{22} \mapsto 3, h_{23} \mapsto 2,$$
$$h_{31} \mapsto 3, h_{32} \mapsto 1, h_{33} \mapsto 2,$$
$$h_{41} \mapsto 3, h_{42} \mapsto 2, h_{43} \mapsto 1\}$$

$$t_3(P_1) \equiv b(y(a)) < b(h_1 \cdot x(a)) \wedge y(a) < h_2 \wedge x(a) - h_3 = y(a)$$

$$MGU = (\{h_1 \mapsto 2, h_2 \mapsto 0, h_3 \mapsto 3\}, \{h_1 \mapsto 4, h_2 \mapsto 3, h_3 \mapsto 7\},$$
$$\{h_1 \mapsto 8, h_2 \mapsto 8, h_3 \mapsto 11\}, \{h_1 \mapsto 16, h_2 \mapsto 15, h_3 \mapsto 15\})$$

$s(1,1\ldots4) = (2,4,8,16)$, $Q(s)$, damit ist $s'(1,i) = 2^i$
$s(2,1\ldots4) = (0,3,8,15)$, $D^2(s)$, damit ist $s'(2,i) = i^2 - 1$
$s(3,1\ldots4) = (3,7,11,15)$, $D(s)$, damit ist $s'(3,i) = 4i - 1$

Für das allgemeine Prädikat $P(i)$ ergibt sich

$$t_1^{-1}(b(y(a)) < b(2^i \cdot x(a)) \wedge y(a) < i^2 - 1 \wedge x(a) - 4i + 1 = y(a))$$
$$\equiv$$
$$b(y) < b(2^i \cdot x) \wedge y < i^2 - 1 \wedge x - 4i + 1 = y$$

Zu überprüfen ist noch, ob $P(0) \equiv P_0$ gilt: $P(0) \equiv b(y) < b(x) \wedge y < -1 \wedge x + 1 = y \equiv P_0$.

4.2.7 Behandlung von Sequenzen nicht strukturgleicher Prädikate

In diesem Abschnitt wird das Problem behandelt, daß in der Sequenz der Prädikate keine strukturgleichen aufeinander folgenden Prädikate vorkommen, was meistens der Fall ist. In diesem Fall kann man versuchen, die Prädikate äquivalent umzuformen, so daß strukturgleiche Prädikate entstehen.

4.2.7.1 Transformation in strukturgleiche Prädikate

Jedes Prädikat P_i der Sequenz *P* wird in DNF umgeformt. Damit sieht *P* so aus:

$$P_0 \equiv D(0,1) \vee \cdots \vee D(0,d(0))$$
$$\vdots$$
$$P_n \equiv D(n,1) \vee \cdots \vee D(n,d(n)) \text{ mit}$$

$D(i,j) \equiv K_1(i,j) \wedge \cdots \wedge K_{k(i,j)}(i,j), i = 1 \ldots n, j = 1 \ldots d(i)$, wobei $d(i)$ die Anzahl der Disjunkte in P_i ist und $k(i,j)$ die Anzahl der Konjunkte in $D(i,j)$. Jedes Konjunkt ist ein Literal. Die Menge der Konjunkte eines Disjunkts wird in *c* Klassen zerlegt, so daß alle Konjunkte einer Klasse strukturgleich sind. Dazu wird die Indexmenge $\{1,\ldots,k(i,j)\}$ der Konjunkte eines Disjunkts in die *c* Mengen $T_t(i,j) \subseteq \{1,\ldots,k(i,j)\}, t = 1 \ldots c$ zerlegt, so daß gilt:

1. $\bigcup_{t=1}^{c} T_t(i,j) = \{1,\ldots,k(i,j)\} \wedge (\forall a,b : 1 \le a,b \le c \wedge a \ne b : T_a(i,j) \cap T_b(i,j) = \varnothing)$ (Partitionierung)
2. $(\forall t : 1 \le t \le c : S(K_k(i,j), k \in T_t(i,j))$ (Strukturgleichheit)

Die Partitionierung ist Gegenstand von Abschnitt 4.2.7.2. Mit dem Verfahren aus den Abschnitten 4.2.5 und 4.2.6 kann dann für jede Klasse *t* von strukturgleichen Konjunkten ein allgemeines Konjunkt $K_t(i,j,m)$ generiert werden, so daß $\bigwedge_{k \in T_t} K_k(i,j) \equiv (\forall m : m = 1 \ldots |T_t(i,j)| : K_t(i,j,m))$ gilt. Da jede Klasse von strukturgleichen Konjunkten eine Menge ist, das Verfahren jedoch auf Sequenzen arbeitet, muß die Menge zuvor in eine Sequenz transformiert werden. Für Klassen mit nur einem Element $K_k(i,j)$ wird $K_t(i,j,m) :\equiv K_k(i,j)$ definiert.

Beispiel (die Konjunkte $K_k(i,j)$ und die Mengen $T_t(i,j)$ werden ohne die Argumente *i* und *j* dargestellt, da sie immer gleich sind):

$$K_1 \equiv 8x > 7, K_2 \equiv b(1), K_3 \equiv 2x > 3, K_4 \equiv y = 0, K_5 \equiv 4x > 5, K_6 \equiv b(2)$$
$$c = 3, T_1 = \{1,3,5\}, T_2 = \{2,6\}, T_3 = \{4\}$$
$$K_1 \wedge K_3 \wedge K_5 \equiv (\forall m : m = 1 \ldots 3 : K_1(m))$$
$$K_2 \wedge K_6 \equiv (\forall m : m = 1 \ldots 2 : K_2(m))$$
$$K_4 \equiv K_3(m)$$
$$\text{mit } K_1(m) \equiv 2^m x > 2m+1, K_2(m) \equiv b(m)$$

•

Nach dieser Transformation gilt (c_j ist die Anzahl der Klassen in Disjunkt $D(i,j)$)

$$D(i,j) \equiv \bigwedge_{t=1}^{c_j} \left(\forall m : m = 1 \ldots |T_t(i,j)| : K_t(i,j,m)\right).$$

Wenn $S(D(i,j), j = 1 \ldots d(i))$ gilt, kann ein allgemeines Disjunkt mit dem Verfahren aus den Abschnitten 4.2.5 und 4.2.6 generiert werden. Damit kann P_i dargestellt werden als

$$(\exists j : 1 \le j \le d(i) : \bigwedge_{t=1}^{c(j)} (\forall m : m = 1 \ldots T(j) : K_t(i, j, m))),$$

wobei $c(j)$ bzw. $T(j)$ die allgemeinen Funktionen der Sequenzen c_j bzw. $|T_t(i, j)|$ sind. Wenn die Sequenz P in dieser Form vorliegt, ist es eher möglich, eine Strukturgleichheit festzustellen.

Beispiel:

$P_0 \equiv false$
$P_1 \equiv x = 0 \wedge b(1)$
$P_2 \equiv b(1) \wedge x = 1 \quad \vee \quad b(2) \wedge x = 2 \wedge b(1)$
$P_3 \equiv b(1) \wedge x = 2 \quad \vee \quad b(2) \wedge x = 3 \wedge b(1) \quad \vee \quad b(2) \wedge x = 4 \wedge b(1) \wedge b(3)$
$P_4 \equiv b(1) \wedge x = 3 \quad \vee \quad b(2) \wedge x = 4 \wedge b(1) \vee b(2) \wedge x = 5 \wedge b(1) \wedge b(3) \quad \vee \quad b(2) \wedge b(4) \wedge x = 6 \wedge b(1) \wedge b(3)$

Man versucht, im letzten, d.h. im Allgemeinen längsten, Prädikat eine Struktur zu finden, und in diesem wieder im längsten Konjunkt $b(2) \wedge b(4) \wedge x = 6 \wedge b(1) \wedge b(3)$. Dieses wird transformiert in $x = 6 \wedge (\forall i{:}1 \le i \le 4{:}b(i))$. Ebenso verfährt man mit den restlichen Konjunkten von P_4 und erhält

$$P_4 \equiv x = 3 \wedge (\forall i : 1 \le i \le 1 : b(i)) \quad \vee \quad x = 4 \wedge (\forall i : 1 \le i \le 2 : b(i)) \vee$$
$$x = 5 \wedge (\forall i : 1 \le i \le 3 : b(i)) \quad \vee \quad x = 6 \wedge (\forall i : 1 \le i \le 4 : b(i))$$

Alle 4 Disjunkte sind strukturgleich, so daß P_4 dargestellt werden kann als

$$(\exists j{:}1 \le j \le 4{:} x = j + 2 \wedge (\forall i{:}1 \le i \le j{:}b(i))).$$

Analog verfährt man mit $P_1 \ldots P_3$ und erhält

$$P_1 \equiv (\exists j{:}1 \le j \le 1{:} x = j - 1 \wedge (\forall i{:}1 \le i \le j{:}b(i)))$$
$$P_2 \equiv (\exists j{:}1 \le j \le 2{:} x = j \wedge (\forall i{:}1 \le i \le j{:}b(i)))$$
$$P_3 \equiv (\exists j{:}1 \le j \le 3{:} x = j + 1 \wedge (\forall i{:}1 \le i \le j{:}b(i)))$$
$$P_4 \equiv (\exists j{:}1 \le j \le 4{:} x = j + 2 \wedge (\forall i{:}1 \le i \le j{:}b(i)))$$

Es gilt $S(P_1, \ldots, P_4)$. Als allgemeines Prädikat erhält man

$$P(k) \equiv (\exists j{:}1 \le j \le k{:} x = j + k - 2 \wedge (\forall i{:}1 \le i \le j{:}b(i)))$$

•

4.2.7.2 Zerlegung in Teilmengen mit Indizes strukturgleicher Konjunkte

Gegeben ist eine Menge von k Konjunkten $\{K_1, K_2, \ldots, K_k\}$. Gesucht sind c Mengen $T_t \subseteq \{1, \ldots, k\}, t = 1 \ldots c$, so daß gilt:

1. $p(k) :\equiv \bigcup_{t=1}^{c} T_t = \{1, \ldots, k\} \wedge (\forall a, b : 1 \le a < b \le c : T_a \cap T_b = \varnothing)$ (Partitionierung) und
2. $sg :\equiv (\forall t : 1 \le t \le c : S(K_k), k \in T_t)$ (Strukturgleichheit)

Aus beweistechnischen Gründen werden noch folgende Prädikate definiert:

$$dj :\equiv (\forall a, b : 1 \le a, b \le c \wedge a \ne b : T_a \cap T_b = \varnothing)$$

$$un(i) :\equiv \bigcup_{t=1}^{c} T_t = \{1,\ldots,i\}$$

$$nc(i) :\equiv i \notin \bigcup_{t=1}^{c} T_t$$

Es gilt $p(i) \equiv un(i) \wedge dj$ (Damit ist die Eigenschaft der Partitionierung in ein von i abhängiges und ein von i unabhängiges Konjunkt zerlegt).
Der folgende Algorithmus führt die Zerlegung durch. Die Grundidee ist, jedes Konjunkt in eine Klasse zu sortieren. Falls es schon eine Klasse mit zu diesem Konjunkt strukturgleichen Konjunkten gibt, wird es in diese Klasse sortiert. Falls nicht, wird eine neue Klasse mit diesem Konjunkt als erstem Element erzeugt. Die (nicht deterministische) Funktion $one\_element\_of(T_j)$ ergibt ein beliebiges Element aus T_j. Falls T_j leer oder noch nicht definiert ist, ist auch das Ergebnis undefiniert ($\perp$). Das Ergebnis der Anwendung des Prädikats *S* auf ein undefiniertes Element ergibt *false*.

```
-- 1 ≤ k
      c := 1; T1 := {};
      --PreF : 1 ≤ k ∧ c = 1 ∧ T1 = {}
      --PostF: sg ∧ dj ∧ un(k)
      --InvF : sg ∧ dj ∧ un(i) ∧ nc(i+1)
      for i in 1 .. k loop
        j := 1;
        if Tj /= {}
  then e := one_element_of(Tj);
  else e := i;
  end if;
      --PreW : sg ∧ dj ∧ un(i-1) ∧ nc(i) ∧ j = 1 ∧
      --PreW : (e in Tj ∨ Tj = {})
      --PostW: sg ∧ dj ∧ un(i-1) ∧ nc(i) ∧ (e in Tj ∨ Tj = {})
      --InvW : sg ∧ dj ∧ un(i-1) ∧ nc(i) ∧ (e in Tj ∨ Tj = {})
      --TermW: c - j
        while ¬S(Ki, Ke) ∧ j ≤ c loop
              j := j + 1;
              e := one_element_of(Tj);
        end loop;
        -- Z: sg ∧ dj ∧ un(i-1) ∧ nc(i) ∧ (e in Tj ∨ Tj = {})
        if S(Ki, Ke)
        then Tj := Tj ∪ { i };
        else c := c + 1;
             Tc := { i };
        end if;
        -- sg ∧ dj ∧ un(i) ∧ nc(i+1)
      end loop;
```

Beweis der Korrektheit:

IF1 := 1. If-Anweisung
IF2 := 2. If-Anweisung
BodyF := Rumpf der For-Schleife
BodyW := Rumpf der While-Schleife

Initialisierung: es gilt $[1 \le k \Rightarrow wp(c := 1; T_1 := \emptyset, 1 \le k \wedge c = 1 \wedge T_1 = \emptyset)]$

For-Schleife:

1. zu zeigen ist $[PreF \Rightarrow sg \wedge dj \wedge un(0) \wedge nc(1)]$

$[PreF \Rightarrow sg \wedge dj \wedge un(0) \wedge nc(1)]$

≡ Definition der Prädikate, Substitution

$[1 \le k \wedge c = 1 \wedge T_1 = \varnothing \Rightarrow S(K_k, k \in \varnothing) \wedge (\forall a,b : 1 \le a,b \le 1 \wedge a \ne b : T_a \cap T_b = \varnothing) \wedge \varnothing = \{1,\ldots,0\} \wedge 1 \notin \varnothing]$

≡ leere Sequenz von Prädikaten ist strukturgleich, ∀ über leeren Bereich

$[1 \le k \wedge c = 1 \wedge T_1 = \varnothing \Rightarrow true \wedge true \wedge true]$

≡ Logik

true

•

2. zu zeigen ist $[sg \wedge dj \wedge un(i-1) \wedge nc(i) \Rightarrow wp(BodyF, sg \wedge dj \wedge un(i) \wedge nc(i+1))]$. Dazu werden 2 Lemmata benötigt.

Lemma 4.22:

$wp(IF1, j = 1 \wedge (e \in T_j \vee T_j = \varnothing) \wedge sg \wedge dj \wedge un(i-1)) \equiv j = 1 \wedge sg \wedge dj \wedge un(i-1)$

Beweis:

$wp(IF1, j = 1 \wedge (e \in T_j \vee T_j = \varnothing) \wedge sg \wedge dj \wedge un(i-1))$

≡ If-Regel, Distributivität

$(T_j \ne \varnothing \wedge (one\_element\_of(T_j) \in T_j \vee T_j = \varnothing) \vee T_j = \varnothing \wedge (i \in T_j \vee T_j = \varnothing)) \wedge$
$j = 1 \wedge sg \wedge dj \wedge un(i-1)$

≡ Mengenlehre, Substitution

$(T_1 \ne \varnothing \wedge (true \vee T_1 = \varnothing) \vee T_1 = \varnothing \wedge (i \in \varnothing \vee \varnothing = \varnothing)) \wedge j = 1 \wedge sg \wedge dj \wedge un(i-1)$

≡ Logik

$(T_1 \ne \varnothing \vee T_1 = \varnothing) \wedge j = 1 \wedge sg \wedge dj \wedge un(i-1)$

≡ Logik

$j = 1 \wedge sg \wedge dj \wedge un(i-1)$

•

Lemma 4.23: $[Z \Rightarrow wp(IF2, sg \wedge dj \wedge un(i) \wedge nc(i+1))]$.

Beweis:

$[Z \Rightarrow wp(IF2, sg \wedge dj \wedge un(i) \wedge nc(i+1))]$

≡ Konjunktivität

$[Z \Rightarrow wp(IF2, sg) \wedge wp(IF2, dj) \wedge wp(IF2, un(i)) \wedge wp(IF2, nc(i+1))]$

Zunächst werden die 4 *wp*s einzeln ermittelt:

$wp(IF2, sg)$

≡ Definition *sg*, If-Regel

$S(K_i, K_e) \wedge wp(T_j := T_j \cup \{i\}, (\forall t : 1 \le t \le c : S(K_k, k \in T_t))) \vee$
$\neg S(K_i, K_e) \wedge wp(c := c+1; T_c := \{i\}, (\forall t : 1 \le t \le c : S(K_k, k \in T_t)))$

≡ ∀-split

$S(K_i, K_e) \wedge wp(T_j := T_j \cup \{i\}, (\forall t : 1 \le t \le c \wedge t \ne j : S(K_k, k \in T_t)) \wedge S(K_k, k \in T_j)) \vee$
$\neg S(K_i, K_e) \wedge wp(c := c+1; T_c := \{i\}, (\forall t : 1 \le t \le c-1 : S(K_k, k \in T_t)) \wedge S(K_k, k \in T_c))$

≡ Zuweisungs-Regel

$$S(K_i,K_e)\wedge(\forall t:1\le t\le c\wedge t\neq j:S(K_k,k\in T_t))\wedge S(K_k,k\in T_j\cup\{i\})\vee$$
$$\neg S(K_i,K_e)\wedge(\forall t:1\le t\le c:S(K_k,k\in T_t))\wedge S(K_k,k\in\{i\})$$

≡ S ist Äquivalenzrelation, $\forall$-split, Mengenlehre

$$S(K_i,K_e)\wedge(\forall t:1\le t\le c:S(K_k,k\in T_t))\wedge(\forall a\in T_j:S(K_i,K_a))\vee$$
$$\neg S(K_i,K_e)\wedge(\forall t:1\le t\le c:S(K_k,k\in T_t))\wedge S(K_i)$$

≡ Definition *sg*, Lemma 4.7 , Distributivität, Logik

$$sg\wedge((\forall a\in T_j:S(K_i,K_a))\vee\neg S(K_i,K_e))$$

•

$$wp(IF2,dj)$$

≡ Definition *dj*, If-Regel

$$S(K_i,K_e)\wedge wp(T_j:=T_j\cup\{i\},(\forall a,b:1\le a,b\le c\wedge a\neq b:T_a\cap T_b=\emptyset))\vee$$
$$\neg S(K_i,K_e)\wedge wp(c:=c+1;T_c:=\{i\},(\forall a,b:1\le a,b\le c\wedge a\neq b:T_a\cap T_b=\emptyset))$$

≡ $\forall$-split

$$S(K_i,K_e)\wedge wp(T_j:=T_j\cup\{i\},$$
$$(\forall a,b:1\le a,b\le c\wedge a\neq b\wedge a\neq j\wedge j\neq b:T_a\cap T_b=\emptyset)\wedge$$
$$(\forall a:1\le a\le c\wedge a\neq j:T_a\cap T_j=\emptyset))\vee$$
$$\neg S(K_i,K_e)\wedge wp(c:=c+1;T_c:=\{i\},$$
$$(\forall a,b:1\le a,b\le c-1\wedge a\neq b:T_a\cap T_b=\emptyset)\wedge(\forall a:1\le a\le c-1:T_a\cap T_c=\emptyset))$$

≡ Zuweisungsregel

$$S(K_i,K_e)\wedge(\forall a,b:1\le a,b\le c\wedge a\neq b\wedge a\neq j\wedge j\neq b:T_a\cap T_b=\emptyset)\wedge$$
$$(\forall a:1\le a\le c\wedge a\neq j:T_a\cap(T_j\cup\{i\}=\emptyset)\vee$$
$$\neg S(K_i,K_e)\wedge(\forall a,b:1\le a,b\le c\wedge a\neq b:T_a\cap T_b=\emptyset)\wedge(\forall a:1\le a\le c:T_a\cap\{i\}=\emptyset)$$

≡ Mengenlehre, Logik, $\forall$-split

$$S(K_i,K_e)\wedge(\forall a,b:1\le a,b\le c\wedge a\neq b:T_a\cap T_b=\emptyset)\wedge(\forall a:1\le a\le c\wedge a\neq j:i\notin T_a)\vee$$
$$\neg S(K_i,K_e)\wedge(\forall a,b:1\le a,b\le c\wedge a\neq b:T_a\cap T_b=\emptyset)\wedge(\forall a:1\le a\le c\wedge a\neq j:i\notin T_a)\wedge i\notin T_j$$

≡ Definition *dj*, Distributivität, Logik

$$dj\wedge(\forall a:1\le a\le c\wedge a\neq j:i\notin T_a)\wedge(S(K_i,K_e)\vee i\notin T_j)$$

•

$$wp(IF2,un(i))$$

≡ Definition *un*, If-Regel

$$S(K_i,K_e)\wedge wp(T_j:=T_j\cup\{i\},\bigcup_{t=1}^{c}T_t=\{1,\ldots,i\})\vee$$
$$\neg S(K_i,K_e)\wedge wp(c:=c+1;T_c:=\{i\},\bigcup_{t=1}^{c}T_t=\{1,\ldots,i\})$$

≡ $\cup$-split

$$S(K_i,K_e) \wedge wp(T_j := T_j \cup \{i\}, T_j \cup \bigcup_{t=1 \wedge t \neq j}^{c} T_t = \{1,\ldots,i\}) \vee$$

$$\neg S(K_i,K_e) \wedge wp(c := c+1; T_c := \{i\}, T_c \cup \bigcup_{t=1}^{c-1} T_t = \{1,\ldots,i\})$$

≡ Zuweisungsregel

$$S(K_i,K_e) \wedge T_j \cup \{i\} \cup \bigcup_{t=1 \wedge t \neq j}^{c} T_t = \{1,\ldots,i\} \vee \neg S(K_i,K_e) \wedge \{i\} \cup \bigcup_{t=1}^{c} T_t = \{1,\ldots,i\}$$

⇐ ∪-split, Mengenlehre

$$S(K_i,K_e) \wedge \bigcup_{t=1}^{c} T_t = \{1,\ldots,i-1\} \vee \neg S(K_i,K_e) \wedge \bigcup_{t=1}^{c} T_t = \{1,\ldots,i-1\}$$

≡ Definition *un*, Distributivität, Logik

$un(i-1)$

•

$wp(IF\,2, nc(i+1))$

≡ Definition *nc*, If-Regel

$$S(K_i,K_e) \wedge wp(T_j := T_j \cup \{i\}, i+1 \notin \bigcup_{t=1}^{c} T_t) \vee$$

$$\neg S(K_i,K_e) \wedge wp(c := c+1; T_c := \{i\}, i+1 \notin \bigcup_{t=1}^{c} T_t)$$

≡ ∪-split

$$S(K_i,K_e) \wedge wp(T_j := T_j \cup \{i\}, i+1 \notin T_j \cup \bigcup_{t=1, t \neq j}^{c} T_t) \vee$$

$$\neg S(K_i,K_e) \wedge wp(c := c+1; T_c := \{i\}, i+1 \notin T_c \cup \bigcup_{t=1}^{c-1} T_t)$$

≡ Zuweisungsregel

$$S(K_i,K_e) \wedge i+1 \notin T_j \cup \{i\} \cup \bigcup_{t=1, t \neq j}^{c} T_t \vee \neg S(K_i,K_e) \wedge i+1 \notin \{i\} \cup \bigcup_{t=1}^{c} T_t$$

≡ ∪-split

$$S(K_i,K_e) \wedge i+1 \notin \{i\} \cup \bigcup_{t=1}^{c} T_t \vee \neg S(K_i,K_e) \wedge i+1 \notin \{i\} \cup \bigcup_{t=1}^{c} T_t$$

≡ Mengenlehre, Definition nc, Distributivität, Logik

$nc(i+1)$

•

Damit gilt

$$wp(IF\,2, sg) \wedge wp(IF\,2, dj) \wedge wp(IF\,2, un(i)) \wedge wp(IF\,2, nc(i+1))$$

⇐

$$sg \wedge ((\forall a \in T_j : S(K_i,K_a)) \vee \neg S(K_i,K_e)) \wedge un(i-1) \wedge nc(i+1) \wedge$$
$$dj \wedge (\forall a \in T_j : 1 \leq a \leq c \wedge a \neq j : i \notin T_a) \wedge (S(K_i,K_e) \vee i \notin T_j)$$

≡ ∧-Subsumierung $un(i-1) \Rightarrow nc(i+1)$, Logik

$$sg \wedge (\forall a \in T_j : S(K_i, K_e) \Rightarrow S(K_i, K_a)) \wedge un(i-1) \wedge$$
$$dj \wedge (\forall a \in T_j : 1 \le a \le c \wedge a \ne j : i \notin T_a) \wedge (i \in T_j \Rightarrow S(K_i, K_e))$$

zu zeigen bleibt

$$[Z \Rightarrow sg \wedge (\forall a \in T_j : S(K_i, K_e) \Rightarrow S(K_i, K_a)) \wedge un(i-1) \wedge$$
$$dj \wedge (\forall a \in T_j : 1 \le a \le c \wedge a \ne j : i \notin T_a) \wedge (i \in T_j \Rightarrow S(K_i, K_e))]$$

≡ Definition Z

$$[sg \wedge dj \wedge un(i-1) \wedge nc(i) \wedge (e \in T_j \vee T_j = \emptyset) \Rightarrow$$
$$sg \wedge (\forall a \in T_j : S(K_i, K_e) \Rightarrow S(K_i, K_a)) \wedge un(i-1) \wedge$$
$$dj \wedge (\forall a \in T_j : 1 \le a \le c \wedge a \ne j : i \notin T_a) \wedge (i \in T_j \Rightarrow S(K_i, K_e))]$$

≡ Logik

$$[sg \wedge dj \wedge un(i-1) \wedge nc(i) \wedge (e \in T_j \vee T_j = \emptyset) \Rightarrow$$
$$(\forall a \in T_j : S(K_i, K_e) \Rightarrow S(K_i, K_a)) \wedge (\forall a \in T_j : 1 \le a \le c \wedge a \ne j : i \notin T_a) \wedge (i \in T_j \Rightarrow S(K_i, K_e))]$$

≡ wegen $[nc(i) \Rightarrow (\forall a \in T_j : 1 \le a \le c \wedge a \ne j : i \notin T_a)]$

$$[sg \wedge dj \wedge un(i-1) \wedge nc(i) \wedge (e \in T_j \vee T_j = \emptyset) \Rightarrow$$
$$(\forall a \in T_j : S(K_i, K_e) \Rightarrow S(K_i, K_a)) \wedge (i \in T_j \Rightarrow S(K_i, K_e))]$$

≡ Logik

$$[(sg \wedge dj \wedge un(i-1) \wedge nc(i) \wedge e \in T_j \Rightarrow$$
$$(\forall a \in T_j : S(K_i, K_e) \Rightarrow S(K_i, K_a)) \wedge (i \in T_j \Rightarrow S(K_i, K_e)))] \wedge$$
$$[(sg \wedge dj \wedge un(i-1) \wedge nc(i) \wedge T_j = \emptyset \Rightarrow$$
$$(\forall a \in T_j : S(K_i, K_e) \Rightarrow S(K_i, K_a)) \wedge (i \in T_j \Rightarrow S(K_i, K_e)))]$$

≡ Mengenlehre

$$[(sg \wedge dj \wedge un(i-1) \wedge nc(i) \wedge T_j = T_j \cup \{e\} \Rightarrow$$
$$(\forall a \in T_j : S(K_i, K_e) \Rightarrow S(K_i, K_a)) \wedge (i \in T_j \Rightarrow S(K_i, K_e)))] \wedge$$
$$[(sg \wedge dj \wedge un(i-1) \wedge nc(i) \wedge T_j = \emptyset \Rightarrow$$
$$(\forall a \in T_j : S(K_i, K_e) \Rightarrow S(K_i, K_a)) \wedge (i \in T_j \Rightarrow S(K_i, K_e)))]$$

≡ Substitution

$$[(sg \wedge dj \wedge un(i-1) \wedge nc(i) \wedge T_j = T_j \cup \{e\} \Rightarrow$$
$$(\forall a \in T_j \cup \{e\} : S(K_i, K_e) \Rightarrow S(K_i, K_a)) \wedge (i \in T_j \cup \{e\} \Rightarrow S(K_i, K_e)))] \wedge$$
$$[(sg \wedge dj \wedge un(i-1) \wedge nc(i) \Rightarrow$$
$$(\forall a \in \emptyset : S(K_i, K_e) \Rightarrow S(K_i, K_a)) \wedge (i \in \emptyset \Rightarrow S(K_i, K_e)))]$$

≡ S ist Äquivalenzrelation, Logik

true

•

Jetzt kann $[sg \wedge dj \wedge un(i-1) \wedge nc(i) \Rightarrow wp(bodyF, sg \wedge dj \wedge un(i) \wedge nc(i+1))]$ gezeigt werden:

$wp(body, sg \wedge dj \wedge un(i))$

≡ Sequenzregel

$wp(j := 1; IF1; WHILE, wp(IF2, sg \wedge dj \wedge un(i)))$

⇐ Lemma 4.23, Monotonie

$wp(j := 1; IF1; WHILE, Z)$

⇐ Monotonie, $PostW \Rightarrow Z$

$wp(j := 1; IF1, j = 1 \wedge (e \in T_j \vee T_j = \emptyset) \wedge sg \wedge dj \wedge un(i-1))$

≡ Lemma 4.22

$wp(j := 1, j = 1 \wedge sg \wedge dj \wedge un(i-1))$

≡ Zuweisungsregel

$sg \wedge dj \wedge un(i-1)$

•

3. es gilt $[sg \wedge dj \wedge un(k) \Rightarrow sg \wedge dj \wedge un(k)]$

4. zu zeigen ist $[1 > k \wedge \mathrm{Pre}F \Rightarrow PostF]$

$[1 > k \wedge 1 \leq k \wedge c = 1 \wedge T_1 = \emptyset \Rightarrow PostF]$

≡ Logik, Arithmetik

$false \Rightarrow PostF$

≡ Logik

true

•

While-Schleife:

1. $[PreW \Rightarrow InvW]$; es gilt

$[sg \wedge dj \wedge un(i-1) \wedge nc(i) \wedge j = 1 \wedge (e \in T_j \vee T_j = \emptyset) \Rightarrow$
$sg \wedge dj \wedge un(i-1) \wedge nc(i) \wedge (e \in T_j \vee T_j = \emptyset)]$

2. $[InvW \wedge \neg(\neg S(K_i, K_e) \wedge j \leq c)) \Rightarrow wp(bodyW, InvW)]$

$[sg \wedge dj \wedge un(i-1) \wedge nc(i) \wedge (e \in T_j \vee T_j = \emptyset) \wedge (S(K_i, K_e) \vee j > c) \Rightarrow$
$wp(j := j+1; e := one\_element\_of(T_j), sg \wedge dj \wedge un(i-1) \wedge nc(i) \wedge (e \in T_j \vee T_j = \emptyset))]$

≡ Zuweisungsregel

$[sg \wedge dj \wedge un(i-1) \wedge nc(i) \wedge (e \in T_j \vee T_j = \emptyset) \wedge (S(K_i, K_e) \vee j > c) \Rightarrow$
$wp(j := j+1, sg \wedge dj \wedge un(i-1) \wedge nc(i) \wedge (one\_element\_of(T_j) \in T_j \vee T_j = \emptyset))]$

≡ Mengenlehre

$[sg \wedge dj \wedge un(i-1) \wedge nc(i) \wedge (e \in T_j \vee T_j = \emptyset) \wedge (S(K_i, K_e) \vee j > c) \Rightarrow$
$wp(j := j+1, sg \wedge dj \wedge un(i-1) \wedge nc(i) \wedge (true \vee T_j = \emptyset))]$

≡ Logik

$[sg \wedge dj \wedge un(i-1) \wedge nc(i) \wedge (e \in T_j \vee T_j = \varnothing) \wedge (S(K_i, K_e) \vee j > c) \Rightarrow$

$wp(j := j+1, sg \wedge dj \wedge un(i-1) \wedge nc(i))]$

≡ wegen $j \notin Var(sg) \cup Var(dj) \cup Var(un(i-1)) \cup Var(nc(i))$

$[sg \wedge dj \wedge un(i-1) \wedge nc(i) \wedge (e \in T_j \vee T_j = \varnothing) \wedge (S(K_i, K_e) \vee j > c) \Rightarrow sg \wedge dj \wedge un(i-1) \wedge nc(i)]$

≡ Logik

true

•

3. $[InvW \wedge \neg(\neg S(K_i, K_e) \wedge j \leq c)) \Rightarrow PostW]$; es gilt

$[sg \wedge dj \wedge un(i-1) \wedge nc(i) \wedge (e \in T_j \vee T_j = \varnothing) \wedge (S(K_i, K_e) \vee j > c) \Rightarrow$

$sg \wedge dj \wedge un(i-1) \wedge nc(i) \wedge (e \in T_j \vee T_j = \varnothing)]$

Die Terminierung der While-Schleife ist leicht zu zeigen

•

Beispiel: $k = 6, S(K_1, K_5, K_6), S(K_2, K_3), S(K_4)$, die Vorbedingung gilt

i	c	T_1	T_2	T_3	j	e
1	1	{}			1	1
		{1}				
2	2		{2}		1	⊥
3					1	1
			{2,3}		2	2
4					1	1
					2	3
	3			{4}	3	⊥
5		{1,5}			1	1
6		{1,5,6}			1	1

•

4.2.7.3 Beispiele

Beispiel 1: Multiplikation nach [AA78: 54]

```
-- x = a ∧ y = b ∧ z = 0 ∧ a ≥ 0 ∧ b ≥ 0
while y /= 0 loop
   if odd(y) then y := y - 1; z := z + x;
   end if;
   y := y / 2; x := 2 * x;
end loop;
-- z = a * b
```

$P_0 \equiv false$

$P_1 \equiv y = 0 \wedge z = ab$

$P_2 \equiv y = 0 \wedge z = ab \vee y = 1 \wedge z + x = ab$

$P_3 \equiv y = 0 \wedge z = ab \vee y = 1 \wedge z + x = ab \vee y = 3 \wedge z + 3x = ab \vee y = 2 \wedge z + 2x = ab$

$$P_4 \equiv y = 0 \wedge z = ab \vee y = 1 \wedge z + x = ab \vee$$
$$y = 3 \wedge z + 3x = ab \vee y = 7 \wedge z + 7x = ab \vee y = 5 \wedge z + 5x = ab \vee$$
$$y = 2 \wedge z + 2x = ab \vee y = 6 \wedge z + 6x = ab \vee y = 4 \wedge z + 4x = ab$$

$$P_5 \equiv y = 0 \wedge z = ab \vee y = 1 \wedge z + x = ab \vee$$
$$y = 3 \wedge z + 3x = ab \vee y = 7 \wedge z + 7x = ab \vee y = 5 \wedge z + 5x = ab \vee$$
$$y = 2 \wedge z + 2x = ab \vee y = 6 \wedge z + 6x = ab \vee y = 4 \wedge z + 4x = ab \vee$$
$$y = 9 \wedge z + 9x = ab \vee y = 15 \wedge z + 15x = ab \vee$$
$$y = 13 \wedge z + 13x = ab \vee y = 11 \wedge z + 11x = ab \vee y = 8 \wedge z + 8x = ab \vee$$
$$y = 12 \wedge z + 12x = ab \vee y = 10 \wedge z + 10x = ab \vee y = 14 \wedge z + 14x = ab$$

$$P_6 \equiv y = 0 \wedge z = ab \vee y = 1 \wedge z + x = ab \vee$$
$$y = 3 \wedge z + 3x = ab \vee y = 7 \wedge z + 7x = ab \vee y = 5 \wedge z + 5x = ab \vee$$
$$y = 2 \wedge z + 2x = ab \vee y = 6 \wedge z + 6x = ab \vee y = 4 \wedge z + 4x = ab \vee$$
$$y = 9 \wedge z + 9x = ab \vee y = 15 \wedge z + 15x = ab \vee$$
$$y = 13 \wedge z + 13x = ab \vee y = 11 \wedge z + 11x = ab \vee y = 8 \wedge z + 8x = ab \vee$$
$$y = 12 \wedge z + 12x = ab \vee y = 10 \wedge z + 10x = ab \vee y = 14 \wedge z + 14x = ab \vee$$
$$y = 16 \wedge z + 16x = ab \vee y = 17 \wedge z + 17x = ab \vee$$
$$y = 19 \wedge z + 19x = ab \vee y = 23 \wedge z + 23x = ab \vee y = 21 \wedge z + 21x = ab \vee$$
$$y = 18 \wedge z + 18x = ab \vee y = 22 \wedge z + 22x = ab \vee y = 20 \wedge z + 20x = ab \vee$$
$$y = 25 \wedge z + 25x = ab \vee y = 31 \wedge z + 31x = ab \vee$$
$$y = 29 \wedge z + 29x = ab \vee y = 27 \wedge z + 27x = ab \vee y = 24 \wedge z + 24x = ab \vee$$
$$y = 28 \wedge z + 28x = ab \vee y = 26 \wedge z + 26x = ab \vee y = 30 \wedge z + 30x = ab$$

Man beginnt mit der Untersuchung von P_6 und stellt fest, daß kein Disjunkt strukturgleiche Konjunkte enthält, d.h. die Klassen strukturgleicher Elemente sind einelementig. Die Disjunkte in P_6 bilden 3 Klassen strukturgleicher Disjunkte, nämlich

$T_1 = \{y = 0 \wedge z = ab\}$, $T_2 = \{y = 1 \wedge z + x = ab\}$ und

$$
\begin{aligned}
T_3 \equiv \{&y = 3 \wedge z + 3x = ab, y = 7 \wedge z + 7x = ab, y = 5 \wedge z + 5x = ab,\\
&y = 2 \wedge z + 2x = ab, y = 6 \wedge z + 6x = ab, y = 4 \wedge z + 4x = ab,\\
&y = 9 \wedge z + 9x = ab, y = 15 \wedge z + 15x = ab,\\
&y = 13 \wedge z + 13x = ab, y = 11 \wedge z + 11x = ab, y = 8 \wedge z + 8x = ab,\\
&y = 12 \wedge z + 12x = ab, y = 10 \wedge z + 10x = ab, y = 14 \wedge z + 14x = ab,\\
&y = 16 \wedge z + 16x = ab, y = 17 \wedge z + 17x = ab,\\
&y = 19 \wedge z + 19x = ab, y = 23 \wedge z + 23x = ab, y = 21 \wedge z + 21x = ab,\\
&y = 18 \wedge z + 18x = ab, y = 22 \wedge z + 22x = ab, y = 20 \wedge z + 20x = ab,\\
&y = 25 \wedge z + 25x = ab, y = 31 \wedge z + 31x = ab,\\
&y = 29 \wedge z + 29x = ab, y = 27 \wedge z + 27x = ab, y = 24 \wedge z + 24x = ab,\\
&y = 28 \wedge z + 28x = ab, y = 26 \wedge z + 26x = ab, y = 30 \wedge z + 30x = ab\}
\end{aligned}
$$

Um ein allgemeines Disjunkt in T_3 zu finden, werden die Disjunkte in T_3 o.B.d.A. nach der Zahl geordnet, die gleich *y* ist. Das ergibt sie Sequenz

$$
\begin{aligned}
T_3 \equiv (&y = 2 \wedge z + 2x = ab, y = 3 \wedge z + 3x = ab, y = 4 \wedge z + 4x = ab,\\
&y = 5 \wedge z + 5x = ab, y = 6 \wedge z + 6x = ab, y = 7 \wedge z + 7x = ab,\\
&y = 8 \wedge z + 8x = ab, y = 9 \wedge z + 9x = ab,\\
&y = 10 \wedge z + 10x = ab, y = 11 \wedge z + 11x = ab, y = 12 \wedge z + 12x = ab,\\
&y = 13 \wedge z + 13x = ab, y = 14 \wedge z + 14x = ab, y = 15 \wedge z + 15x = ab,\\
&y = 16 \wedge z + 16x = ab, y = 17 \wedge z + 17x = ab,\\
&y = 18 \wedge z + 18x = ab, y = 19 \wedge z + 19x = ab, y = 20 \wedge z + 20x = ab,\\
&y = 21 \wedge z + 21x = ab, y = 22 \wedge z + 22x = ab, y = 23 \wedge z + 23x = ab,\\
&y = 24 \wedge z + 24x = ab, y = 25 \wedge z + 25x = ab,\\
&y = 26 \wedge z + 26x = ab, y = 27 \wedge z + 27x = ab, y = 28 \wedge z + 28x = ab,\\
&y = 29 \wedge z + 29x = ab, y = 30 \wedge z + 30x = ab, y = 31 \wedge z + 31x = ab)
\end{aligned}
$$

Das allgemeine Prädikat der Sequenz T_3 ergibt sich zu $(\exists i{:}2 \le i \le 31{:}\, y = i \wedge z + ix = ab)$. Damit läßt sich P_6 darstellen als

$$P_6 \equiv y = 0 \wedge z = ab \vee y = 1 \wedge z + x = ab \vee (\exists i{:}2 \le i \le 31{:}\, y = i \wedge z + ix = ab)$$

Analog ergibt sich für die anderen Prädikate

$$
\begin{aligned}
P_0 &\equiv \mathit{false}\\
P_1 &\equiv y = 0 \wedge z = ab\\
P_2 &\equiv y = 0 \wedge z = ab \vee y = 1 \wedge z + x = ab\\
P_3 &\equiv y = 0 \wedge z = ab \vee y = 1 \wedge z + x = ab \vee (\exists i{:}2 \le i \le 3{:}\, y = i \wedge z + ix = ab)\\
P_4 &\equiv y = 0 \wedge z = ab \vee y = 1 \wedge z + x = ab \vee (\exists i{:}2 \le i \le 7{:}\, y = i \wedge z + ix = ab)\\
P_5 &\equiv y = 0 \wedge z = ab \vee y = 1 \wedge z + x = ab \vee (\exists i{:}2 \le i \le 15{:}\, y = i \wedge z + ix = ab)
\end{aligned}
$$

Es gilt $S(P_3,\ldots,P_6)$ mit dem allgemeinen Prädikat

$$P(j) \equiv y = 0 \wedge z = ab \vee y = 1 \wedge z + x = ab \vee (\exists i{:}2 \le i \le 2^{j-1} - 1{:}\, y = i \wedge z + ix = ab)$$

Das ist durch Induktion über *j* zu beweisen, wobei $j = 3$ den Induktionsanfang bildet.

$j = 3:\ P(3) \equiv y = 0 \wedge z = ab \vee y = 1 \wedge z + x = ab \vee (\exists i{:}2 \le i \le 2^{3-1} - 1{:}\, y = i \wedge z + ix = ab) \equiv P_3$

$j \to j+1$: zu zeigen ist

$P(j+1) \equiv y = 0 \wedge z = ab \vee y = 1 \wedge z + x = ab \vee (\exists i{:}2 \le i \le 2^{j} - 1{:}\, y = i \wedge z + ix = ab)$

$P(j+1)$

≡ Definition von P

$f^{j+1}(false)$

≡

$f(f^{j}(false))$

≡ Definition von P

$f(P(j))$

≡ Induktionsvoraussetzung

$f(y = 0 \wedge z = ab \vee y = 1 \wedge z + x = ab \vee (\exists i{:}2 \le i \le 2^{j-1} - 1{:}\, y = i \wedge z + ix = ab))$

≡ Definition von f

$y = 0 \wedge z = ab \vee$

$y \neq 0 \wedge wp(S, y = 0 \wedge z = ab \vee y = 1 \wedge z + x = ab \vee (\exists i{:}2 \le i \le 2^{j-1} - 1{:}\, y = i \wedge z + ix = ab))$

≡ Zuweisungsregel

$y = 0 \wedge z = ab \vee$

$y \neq 0 \wedge wp(IF, \left\lfloor \frac{y}{2} \right\rfloor = 0 \wedge z = ab \vee \left\lfloor \frac{y}{2} \right\rfloor = 1 \wedge z + 2x = ab \vee (\exists i{:}2 \le i \le 2^{j-1} - 1{:} \left\lfloor \frac{y}{2} \right\rfloor = i \wedge z + 2ix = ab))$

≡ If-Regel

$y = 0 \wedge z = ab \vee y \neq 0 \wedge ($

$odd(y) \wedge (\left\lfloor \frac{y-1}{2} \right\rfloor = 0 \wedge z = ab \vee \left\lfloor \frac{y-1}{2} \right\rfloor = 1 \wedge z + 3x = ab \vee$

$(\exists i : 2 \le i \le 2^{j-1} - 1 : \left\lfloor \frac{y-1}{2} \right\rfloor = i \wedge z + (2i+1)x = ab))$

$\vee \neg odd(y) \wedge (\left\lfloor \frac{y}{2} \right\rfloor = 0 \wedge z = ab \vee \left\lfloor \frac{y}{2} \right\rfloor = 1 \wedge z + 2x = ab \vee (\exists i : 2 \le i \le 2^{j-1} - 1 : \left\lfloor \frac{y}{2} \right\rfloor = i \wedge z + 2ix = ab)))$

≡ Arithmetik

$y = 0 \wedge z = ab \vee y = 1 \wedge z = ab \vee$

$y = 3 \wedge z + 3x = ab \vee (\exists i{:}2 \le i \le 2^{j-1} - 1{:}\, y = 2i + 1 \wedge z + (2i+1)x = ab)$

$\vee\, y = 2 \wedge z + 2x = ab \vee (\exists i{:}2 \le i \le 2^{j-1} - 1{:}\, y = 2i \wedge z + 2ix = ab))$

≡ Arithmetik, Logik

$y = 0 \wedge z = ab \vee y = 1 \wedge z = ab \vee (\exists i{:}2 \le i \le 2^{j} - 1{:}\, y = i \wedge z + ix = ab))$

•

Damit ist die schwächste Vorbedingung der Schleife

$y = 0 \wedge z = ab \vee y = 1 \wedge z + x = ab \vee (\exists j: j \geq 3: (\exists i: 2 \leq i \leq 2^{j-1} - 1: y = i \wedge z + ix = ab))$

≡ Arithmetik, Logik

$y = 0 \wedge z = ab \vee y = 1 \wedge z + x = ab \vee (\exists i: i \geq 2: y = i \wedge z + ix = ab)$

≡ Arithmetik, Logik

$(\exists i: i \geq 0: y = i \wedge z + ix = ab)$

Denn es gilt $(\exists i: i \geq 2: y = i \wedge z + ix = ab) \equiv (\exists j: j \geq 3: (\exists i: 2 \leq i \leq 2^{j-1} - 1: y = i \wedge z + ix = ab))$.
Beweis:
⇒:
$(\exists i: i \geq 2 \wedge y = i \wedge z + ix = ab) \Rightarrow (\exists j: j \geq 3 \wedge (\exists i: 2 \leq i \leq 2^{j-1} - 1 \wedge y = i \wedge z + ix = ab))$

≡ Skolemisierung, gebundene Umbenennung

$(\forall i: i \geq 2 \wedge y = i \wedge z + ix = ab \Rightarrow (\exists j: j \geq 3 \wedge (\exists k: 2 \leq k \leq 2^{j-1} - 1 \wedge y = k \wedge z + kx = ab)))$

≡ mit $k = i$

$(\forall i: i \geq 2 \wedge y = i \wedge z + ix = ab \Rightarrow (\exists j: j \geq 3 \wedge 2 \leq i \leq 2^{j-1} - 1 \wedge y = i \wedge z + ix = ab))$

≡ mit $j = i + 1$

$(\forall i: i \geq 2 \wedge y = i \wedge z + ix = ab \Rightarrow i + 1 \geq 3 \wedge 2 \leq i \leq 2^{i} - 1 \wedge y = i \wedge z + ix = ab)$

≡

true

⇐:
$(\exists j: j \geq 3 \wedge (\exists i: 2 \leq i \leq 2^{j-1} - 1 \wedge y = i \wedge z + ix = ab)) \Rightarrow (\exists i: i \geq 2 \wedge y = i \wedge z + ix = ab)$

≡ gebundene Umbenennung

$(\exists j, i: j \geq 3 \wedge 2 \leq i \leq 2^{j-1} - 1 \wedge y = i \wedge z + ix = ab) \Rightarrow (\exists k: k \geq 2 \wedge y = k \wedge z + kx = ab)$

≡ Skolemisierung

$(\forall j, i: j \geq 3 \wedge 2 \leq i \leq 2^{j-1} - 1 \wedge y = i \wedge z + ix = ab \Rightarrow (\exists k: k \geq 2 \wedge y = k \wedge z + kx = ab))$

≡ mit $k = i$

$(\forall j, i: j \geq 3 \wedge 2 \leq i \leq 2^{j-1} - 1 \wedge y = i \wedge z + ix = ab \Rightarrow i \geq 2 \wedge y = i \wedge z + ix = ab)$

≡

true

•

Jetzt kann die Korrektheit der Schleife gezeigt werden. Es gilt

$[x = a \wedge y = b \wedge z = 0 \wedge a \geq 0 \wedge b \geq 0 \Rightarrow (\exists i : i \geq 0 : y = i \wedge z + ix = ab)]$

≡ Substitution

$[a \geq 0 \wedge b \geq 0 \Rightarrow (\exists i : i \geq 0 : b = i \wedge 0 + ia = ab)]$

≡ mit $i = b$

$[a \geq 0 \wedge b \geq 0 \Rightarrow b \geq 0 \wedge ba = ab]$

$\equiv$

true

•

Jetzt werden 2 kleine Beispiele nicht korrekter Schleifen gezeigt, einmal eine terminierende nicht korrekte Schleife und einmal eine nicht terminierende Schleife.

Beispiel 2: Nicht terminierende Schleife

```
-- i = 1
while i > 0 loop
   i := i+1;
end loop;
-- i ≤ 0
```

$P_0 \equiv false$

$P_1 \equiv i \leq 0$

$P_2 \equiv i \leq 0 \vee i > 0 \wedge i + 1 \leq 0 \equiv i \leq 0$

Damit ist $wp(LOOP, i \leq 0) \equiv i \leq 0$ und die Verifikationsbedingung ist $[i > 0 \Rightarrow i \leq 0]$, was nicht gilt, d.h. die Schleife ist nicht korrekt. An diesem Beispiel sieht man auch, daß die Eigenschaft Terminierung keine Eigenschaft der Schleife allein ist, sondern eine Eigenschaft des Paares (Vorbedingung, Schleife).

•

Beispiel 3: Nicht korrekte, terminierende Schleife

```
-- i = 1
while i > 0 loop
   i := i-1;
end loop;
-- i < 0
```

$P_0 \equiv false$

$P_1 \equiv i < 0$

$P_2 \equiv i < 0 \vee i > 0 \wedge i - 1 < 0 \equiv i < 0$

Damit ist $wp(LOOP, i < 0) \equiv i < 0$ und die Verifikationsbedingung ist $[i = 1 \Rightarrow i < 0]$, was nicht gilt, d.h. die Schleife ist nicht korrekt.

•

Beispiel 4: Ganzzahlige Division mit Rest (siehe S. 108)

$P_0 \equiv false$

$P_1 \equiv 0 \leq r < y \wedge qy + r = x$

$P_2 \equiv 0 \leq r < 2y \wedge qy + r = x$

$P_3 \equiv 0 \leq r < 3y \wedge qy + r = x$

$P_4 \equiv 0 \leq r < 4y \wedge qy + r = x$

$P_5 \equiv 0 \leq r < 5y \wedge qy + r = x$

$\min\{i | S(P_i, P_{i+1})\} = 2$, also beginnt die betrachtete Sequenz bei P_2.

$t_1(P_2) \equiv 0 \leq r(a) < 2y(a) \wedge q(a)y(a) + r(a) = x(a)$

$t_1(P_3) \equiv 0 \leq r(a) < 3y(a) \wedge q(a)y(a) + r(a) = x(a)$

$t_1(P_4) \equiv 0 \leq r(a) < 4y(a) \wedge q(a)y(a) + r(a) = x(a)$

$t_1(P_5) \equiv 0 \leq r(a) < 5y(a) \wedge q(a)y(a) + r(a) = x(a)$

$t(P_2) \equiv 1 \leq r(a) < 2y(a) \wedge q(a)y(a) + r(a) = x(a)$

$t(P_3) \equiv h_{31} \leq r(a) < h_{32}y(a) \wedge q(a)y(a) + r(a) = x(a)$

$$t(P_4) \equiv h_{41} \le r(a) < h_{42} y(a) \wedge q(a) y(a) + r(a) = x(a)$$
$$t(P_5) \equiv h_{51} \le r(a) < h_{52} y(a) \wedge q(a) y(a) + r(a) = x(a)$$

Es gilt $U(t(P_2,\ldots,P_5))$.

$$t_3(P_2) \equiv h_1 \le r(a) < h_2 y(a) \wedge q(a) y(a) + r(a) = x(a) .$$
$$MGU = (\{h_1 \mapsto 0, h_2 \mapsto 2\}, \{h_1 \mapsto 0, h_2 \mapsto 3\}, \{h_1 \mapsto 0, h_2 \mapsto 4\}, \{h_1 \mapsto 0, h_2 \mapsto 5\})$$

$s(1,2\ldots5) = (0,0,0,0)$, $D^0(s)$, damit ist $s'(1,i) = 0$
$s(2,2\ldots5) = (2,3,4,5)$, $D(s)$, damit ist $s'(2,i) = i$

Das allgemeine Prädikat ist $P(i) \equiv 0 \le r < iy \wedge qy + r = x$. Der Beweis, daß $P(i)$ der induktiven Definition von P_i genügt, wurde schon in Abschnitt 4.2.2 geführt, ebenso die Ermittlung der schwächsten Vorbedingung und der Korrektheitsbeweis der Schleife.

5 Falsifikationsbedingungen

Das Ziel der Programmentwicklung ist ein korrektes Programm. Die Korrektheit wird durch den Beweis einer VC gezeigt. Doch falls die Korrektheit während der Entwicklung nicht gezeigt werden kann, ist man daran interessiert zu wissen, ob und warum das Programm nicht korrekt ist. (Die beiden Aussagen *die Korrektheit kann nicht gezeigt werden* und *das Programm ist nicht korrekt* sind nicht äquivalent!). In Analogie zur VC, die die Korrektheit impliziert, ist eine Bedingung, die die Nichtkorrektheit impliziert, eine *Falsifikationsbedingung FC*. In diesem Kapitel wird zunächst die automatische Erzeugung von FCs dargestellt, was relativ einfach ist, da eine FC leicht aus einer VC hergeleitet werden kann (Abschnitt 5.1). Das Ziel ist jedoch eine möglichst einfache FC, d.h. ein konkreter Anfangszustand, der eine Zusicherung verletzt (Abschnitt 5.7). Durch die Lokalisierung der verletzten Zusicherung bekommt man einen sehr guten Hinweis darauf, wo ein Programmfehler liegen kann. Dieses Problem läßt sich auf ein *constraint programming problem* bzw. *integer programming problem ipp* (wenn man es, wie in dieser Arbeit, nur mit ganzen Zahlen zu tun hat) reduzieren. Auch das Problem der automatischen Datenabhängigkeitsanalyse von Programmen mit Arrayzugriffen läßt sich auf ein solches Problem reduzieren (mit einigen kleinen Unterschieden, Abschnitt 5.1). Daher werden zur Lösung des Problems, einen konkreten Anfangszustand, der zu einer verletzten Zusicherung führt, zu finden, Methoden der automatischen Datenabhängigkeitsanalyse verwendet (Abschnitte 5.2-5.6). Einige Beispiele erläutern (Beispiele 1-6) bzw. zeigen die Begrenztheit des Verfahrens (Beispiele 7 und 8). Neu sind dabei nicht die verwendeten Methoden, sondern die Erzeugung der FC und Vereinfachung der FC durch Reduktion des Problems auf ein anderes Problem, das mit bekannten Verfahren gelöst wird.

Bei der automatischen Verifikation von Programmen mit Spezifikation kommt es während der Entwicklung vor, daß eine VC nicht bewiesen werden kann. Das kann prinzipiell 2 verschiedene Ursachen haben.

1. Die VC ist nicht gültig, d.h. Programm und Spezifikation passen nicht zusammen.
2. Die VC ist zwar gültig, aber kann vom Beweiser nicht bewiesen werden, was wiederum verschiedene Gründe haben kann.

Falls Programm und Spezifikation nicht zusammenpassen, wird in dieser Arbeit davon ausgegangen, daß das Programm nicht korrekt ist. Es ist oft sehr schwierig zu ermitteln, was im Programm falsch ist. Aber um zu zeigen, daß ein Programm nicht korrekt ist, genügt schon ein Gegenbeispiel. Damit ist ein Anfangszustand gemeint, der während der Ausführung des Programmes zu einem Zustand führt, der eine Zusicherung verletzt.

Beispiel:

```
subtype t is integer range 0..100;
x, y: t;
-- 0 ≤ x ≤ 100 ∧ 0 ≤ y ≤ 100 ∧ x = a ∧ y = b
x := x + y;
y := x - y;
x := x - y;
-- 0 ≤ x ≤ 100 ∧ 0 ≤ y ≤ 100 ∧ x = b ∧ y = a
```

Die VC lautet

$$[0 \le x \le 100 \wedge 0 \le y \le 100 \wedge x = a \wedge y = b \Rightarrow$$
$$0 \le y \le 100 \wedge 0 \le x + y - y \le 100 \wedge y = b \wedge x = a \wedge 0 \le x + y \le 100]$$

Sie kann vom Beweiser nicht bewiesen werden. Falls das Programm nicht korrekt ist, muß es einen Anfangszustand geben, der zu einem Zustand führt, der eine Zusicherung verletzt. So ein Anfangszustand ist z.B. $x = 100 \wedge y = 1$. Also ist das Programm nicht korrekt.

In diesem Kapitel wird ein partielles Verfahren angegeben, mit dem aus einer nicht beweisbaren VC automatisch ein Gegenbeispiel erzeugt werden kann. Partiell heißt, daß das Verfahren nicht immer aus einer nicht beweisbaren VC ein Gegenbeispiel erzeugen kann. Das Verfahren arbeitet nach folgendem Algorithmus:

```
function generate_counterexample (VC: predicate)
return result_type is
 FC:       constant predicate := generate_FC(VC);
 solution: constant solution_type := sol(FC);
 ce:       solution_type;
begin
 if solution = {} then return no_success; end if;
 ce := generate_counterexample(solution);
 if exact(VC)
then return ce;
 elsif
      there is a violation of an assertion during execution of
     the program with initial values ce
     then return ce;
     else return no_success;
     end if;
 end if;
end generate_counterexample;
```

Der Grund für die Unterscheidung zwischen einer FC, die aus einer exakten und einer FC, die aus einer approximierten VC erzeugt wurde, wird im nächsten Abschnitt (Abschnitt 5.1) angegeben.

Verwendete Typen und Funktionen

`predicate`	ein geeigneter Typ für Prädikate
`result_type`	ein Typ bestehend aus Variablenbelegungen und dem Literal `no_success`
`solution_type`	ein geeigneter Typ für eine Variablenbelegung
`function generate_FC(VC: predicate) return predicate`	eine Funktion, die aus einer nicht beweisbaren VC eine FC generiert, siehe Definition 5.1
`function sol(FC: predicate) return solution_type`	eine Funktion zur Ermittlung einer Variablenbelegung, siehe Abschnitte 5.2-5.6
`function generate_counterexample (solution: solution_type) return solution_type`	eine Funktion zur Ermittlung eines Gegenbeispiels aus einer Lösung der FC, siehe Abschnitt 5.7
`function exact(VC: predicate) return boolean`	diese Funktion ergibt genau dann *true*, wenn die VC eine exakte VC ist

Das Verfahren hat folgende Eigenschaft:
Falls ein Gegenbeispiel gefunden wird, dann ist das Programm nicht korrekt. Falls keines gefunden wird, kann es sein, daß es keines gibt oder daß es wegen der Begrenztheit der Verfahrens nicht gefunden werden kann. Man kann also folgende Fälle unterscheiden:

1. VC kann bewiesen werden ⇒ Programm korrekt (bewiesen)
2. VC kann nicht bewiesen werden ∧ Gegenbeispiel gefunden ⇒ Programm nicht korrekt (widerlegt)
3. VC kann nicht bewiesen werden ∧ kein Gegenbeispiel gefunden ⇒ ??? (weder bewiesen noch widerlegt)

Das Finden eines Gegenbeispiels erfüllt also 2 Zwecke:

1. Beweis, daß ein Programm nicht korrekt ist
2. Hilfe bei der Beantwortung der Frage „was / wo / warum nicht korrekt?“

5.1 Formalisierung des Problems

Alle VCs sind Implikationen, die im Laufe des Beweises in eine Konjunktion von *k* Klauseln K_i umgeformt werden, also $VC \equiv \bigwedge_{i=1}^{k} K_i$. Falls eine VC nicht bewiesen werden kann, heißt das, daß mindestens eine Klausel $K_a, 1 \le a \le k,$ nicht bewiesen werden kann. Jede Klausel und damit auch K_a hat die Form $L_1 \wedge \cdots \wedge L_n \Rightarrow L$, wobei die L_i bzw. *L* Literale sind. Alle Literale enthalten freie Variablen. (Falls ein Literal keine freien Variablen enthält, hat es unabhängig von allen anderen einen Wahrheitswert, der für dieses Literal substituiert werden kann. Falls ein $L_i \equiv true$, so kann es eliminiert werden. Falls ein $L_i \equiv false$, so ist die ganze Klausel *true*. Falls $L \equiv true$, so ist die ganze Klausel *true*. Falls $L \equiv false$, so kann eine neue äquivalente Klausel ohne *L* erzeugt werden.)
Die freien Variablen sind allquantifiziert, also $(\forall x : L_1 \wedge \cdots \wedge L_n \Rightarrow L)$, wobei *x* die freien Variablen in der Klausel bezeichnet. Da K_a nicht beweisen werden kann, geht man davon aus, daß sie nicht gültig ist, daß also $\neg(\forall x : L_1 \wedge \cdots \wedge L_n \Rightarrow L)$ gilt. Das ist äquivalent zu $(\exists x : L_1 \wedge \cdots \wedge L_n \wedge \neg L)$.

Definition 5.1: Eine *Falsifikationsbedingug FC* zu einer nicht beweisbaren exakten VC ist definiert als $FC :\equiv \neg K_a$, wobei K_a eine nicht beweisbare Klausel der VC ist. Die Lösungsmenge $Sol(FC)$ einer FC der Form $(\exists x : L_1 \wedge \cdots \wedge L_n \wedge \neg L)$ ist definiert als $Sol(FC) := \{x \mid L_1 \wedge \cdots \wedge L_n \wedge \neg L \wedge found(x)\}$. Dabei bedeutet *found*(*x*), daß das in diesem Kapitel beschriebene Verfahren (Abschnitte 5.2-5.6) die Werte *x* berechnen konnte. Der Grund dafür, daß die VC exakt sein muß, wird weiter unten angegeben.

•

Lemma 5.2: $[VC \Rightarrow \neg FC]$.
Beweis:

$[VC \Rightarrow \neg FC]$

≡ Form von VC, Definition FC

$[\bigwedge_{i=1}^{k} K_i \Rightarrow \neg(\neg K_a)]$

≡ Logik

true

•

Lemma 5.3: $[Sol(FC) \neq \varnothing \Rightarrow FC]$.
Beweis: falls das Verfahren eine Lösung findet, hat die FC eine Lösung, d.h. ist gültig. Das Problem ist, daß die umgekehrte Richtung im Allgemeinen nicht gilt. D.h. falls das Verfahren keine Lösung findet, kann man daraus nicht schließen, daß die FC keine Lösung hat. Das liegt an der Begrenztheit des Verfahrens.

•

Damit ist klar, daß $Sol(FC) = \varnothing$ keine VC ist. $Sol(FC) = \varnothing$ wäre nur dann eine VC, wenn auch die Umkehrungen von Lemma 5.2 und Lemma 5.3 gelten würden. (Denn damit hätte

man $[Sol(FC) = \emptyset \Rightarrow \neg FC]$ und mit Lemma 5.2 $[Sol(FC) = \emptyset \Rightarrow VC]$ und mit der Definition von VC und der Transitivität bekäme man $[Sol(FC) = \emptyset \Rightarrow Korrektheit]$).
Außerdem impliziert $Sol(FC) \neq \emptyset$ die Nichtkorrektheit nur, wenn $Sol(FC)$ aus einer exakten VC erzeugt wird, wie Definition 5.1, d. h. es gilt

Lemma 5.4: Falls $Sol(FC)$ aus einer exakten VC erzeugt wird, gilt

$$[Sol(FC) \neq \emptyset \Rightarrow \neg Korrektheit].$$

Beweis:

$Sol(FC) \neq \emptyset$

$\Rightarrow$ Lemma 5.3

FC

$\Rightarrow$ Lemma 5.2

$\neg$VC

$\equiv$ Definition *exakte VC*

$\neg$Korrektheit

•

Bei approximierten VCs könnte man statt des letzten $\equiv$ im Beweis nur ein $\Leftarrow$ nehmen und der Beweis wäre nicht durchführbar. Dennoch kann man mit einer aus einer approximierten VC gewonnen FC arbeiten. Dazu wird aus der FC ein Anfangszustand ermittelt und mit diesem das Programm ausgeführt. Wenn dann eine Zusicherung verletzt wird, ist bewiesen, daß das Programm nicht korrekt ist (siehe auch Abschnitt 5.8 und die Beispiele in Abschnitt 5.8).
Das Problem, ein Gegenbeispiel zu finden, kann auf ein *constraint programming problem* cpp [C97: 2066-2093] zurückgeführt werden. Auch das Problem der automatischen Bestimmung von Datenabhängigkeiten von Arrayzugriffen in geschachtelten For-Schleifen kann auf ein cpp zurückgeführt werden [MHL91]. Zur Lösung des Problems, ein Gegenbeispiel zu finden, kann man daher versuchen, die gleichen Methoden anzuwenden, wie bei letzterem Problem.
So wie die Gültigkeit eines Satzes im Allgemeinen nicht entscheidbar ist, ist auch das cpp nicht entscheidbar. Wenn man sich im Bereich der ganzen Zahlen aber auf die Presburger-Arithmetik [D91: 108] beschränkt, so ist das Problem entscheidbar, aber mit einem Aufwand, der im schlechtesten Fall von der Größenordnung $2^{2^{2^{O(n)}}}$ ist [O78]. Für das Beispiel ergibt sich das Problem

$$(\exists x, y, a, b: 0 \leq x \leq 100 \wedge 0 \leq y \leq 100 \wedge x = a \wedge y = b \wedge x + y > 100)$$

mit der Lösung $x = 100 \wedge y = 1 \wedge a = 100 \wedge b = 1$. In diesem Fall hat man nur einen äußeren Existenzquantor. Probleme dieser Art gehören zur Klasse *linear integer programming* lipp [W91]. Dafür gibt es relativ effiziente Algorithmen. Diese Klasse stellt jedoch eine ziemlich starke Einschränkung dar. Die Einschränkungen sind:

1. Nur ein äußerer Existenzquantor erlaubt (siehe 5.5)
2. Nur Konjunktionen von constraints
3. Constraints sind lineare Gleichungen oder Ungleichungen (siehe 5.3)
4. Keine uninterpretierten Funktionssymbole (Arrays) (siehe 5.4)
5. Nur folgende Funktionen und Prädikate sind erlaubt: +, -, ·, =, ≤, ≥, ∧, ∨, ¬ (siehe 5.2)

Bemerkung: die Standarddefinition eines ipp ist:

gesucht ist das größte *x*, das die constraints erfüllt, wobei *x* ein ganzzahliger Vektor ist. Das hier auftauchende Problem, nur irgendein *x* zu finden, scheint auf den ersten Blick einfacher zu sein, doch nach [MHL91] haben beide Probleme gleiche Komplexität.

•

Das Verfahren zur Lösung dieses Problems besteht im Wesentlichen in der Transformation auf ein lipp, das mit Hilfe des Ω-Tests von Pugh und Wonnacott [P92] gelöst wird. Dabei wird jede der 5 Einschränkungen nacheinander eliminiert, und zwar im Wesentlichen mit den Methoden, die von Pugh und Wonnacott im Bereich der Datenabhängigkeitsanalyse entwikkelt wurden. Nur die 2. und 5. Einschränkung kommen bei der Datenabhängigkeitsanalyse nicht vor (da sie sowieso erfüllt sind), können aber leicht gelöst werden. Die Eingabe des Verfahrens ist ein ipp, die Ausgabe ist ein transformiertes Prädikat $Sol(ipp)$. In *Idealform* stellt $Sol(ipp)$ eine Variablenbelegung bzw. einen konkreten Zustand dar, ist also von der Form $\bigwedge_{i=1}^{n} v_i = w_i$, wobei v_i die in $Sol(ipp)$ vorkommenden Variablen sind und $w_i \in Type(v_i)$. So eine konkrete Variablenbelegung ist im Allgemeinen nur eine von mehreren Lösungen. Es kann vorkommen, daß durch Einführung von Hilfsvariablen im Laufe des Verfahrens das Prädikat $Sol(ipp)$ mehr Variablen hat als das ursprüngliche ipp, also $free(Sol(ipp)) \supset free(ipp)$.

5.2 Repräsentation von Prädikaten und Funktionen

Prädikate und Funktionen, die nicht Einschränkung 5 genügen, werden so transformiert, daß sie Einschränkung 5 genügen.

$$a < b \equiv a \le b - 1$$
$$a > b \equiv a \ge b + 1$$
$$a \ne b \equiv a \ge b + 1 \vee a \le b - 1$$
$$L(E / F) \equiv (\exists t, a: L(t) \wedge tF + a = E \wedge 0 \le a \le F - 1),$$
wobei E und F lineare Ausdrücke sind
$$m = (MAX\ \ j: 1 \le j \le n: b(j)) \equiv (\forall j: 1 \le j \le n: m \ge b(j)) \wedge (\exists j: 1 \le j \le n: m \ge b(j))$$

Andere Prädikate (z.B. *prim*) und Funktionen (z.B. *abs, ggt*) müssen entsprechend transformiert werden.

5.3 Affinisierung bestimmter polynomialer constraints

Lineare constraints stellen eine starke Einschränkung dar. Damit ist z.B. schon die Multiplikation zweier Variablen ausgeschlossen. Zur Linearisierung wird die Methode aus [MP94] verwendet. Sie geht folgendermaßen vor: ein polynomialer constraint wird ein eine äquivalente Konjunktion linearer constraints und eines polynomialen der folgenden Form zerlegt:

$xy \ge c$	hyperbolische Ungleichung
$xy = c$	hyperbolische Gleichung
$ax^2 + by^2 \ge c$	elliptische Ungleichung
$ax^2 + by^2 = c$	elliptische Gleichung

(Dabei sind *x* und *y* Variablen und *a*, *b* und $c \in \mathbb{Z}$)

Das polynomiale constraint wird affinisiert. Dazu werden zunächst die Relationen =, <, >, ≠ und ≤ durch ≥ ersetzt (Abschnitt 5.2). Der Algorithmus wendet dann die Schritte Faktorisierung (Abschnitt 5.3.1) und Linearisierung (Abschnitt 5.3.2) an, solange es geht. D.h. jeder dieser Schritte verringert den Grad des constraints um 1 oder 0. Der Algorithmus terminiert,

wenn alle constraints linear sind (mit Erfolg) oder keine Verringerung des Grades mehr erfolgt (kein Erfolg).

5.3.1 Faktorisierung

Ein constraint der Form $\sum_{i=1}^{n} a_i x R_i \geq c$ wird transformiert in $(\exists y: y = \sum_{i=1}^{n} a_i R_i \wedge xy \geq c)$. Dabei sind a_i und c Zahlen, x eine Variable und R_i ein Produkt von Variablen oder 1. (*common term*).

Ein constraint der Form $a_x^2 x^2 + b_x x - a_y^2 y^2 - b_y y + c \geq 0$ wird transformiert in

$$(\exists m,n: m = 2a_x^2 a_y x - 2a_y^2 a_x y + b_x a_y - b_y a_x \wedge n = 2a_x^2 a_y x + 2a_y^2 a_x y + b_x a_y + b_y a_x \wedge mn \geq a_y^2 b_x^2 - a_x^2 b_y^2 - 4a_x^2 a_y^2 c)$$

.

Dabei sind $a_x > 0$, $a_y > 0$, b_x, b_y und c Zahlen und x und y Variablen. Falls der Koeffizient von x^2 keine Quadratzahl ist, muß der ganze Term mit einer Zahl multipliziert werden, so daß der Koeffizient von x^2 eine Quadratzahl ist. Falls danach der Koeffizient von y^2 keine Quadratzahl ist, ist Faktorisierung nicht möglich. (*breaking quadratic equation*)

Ein constraint der Form $a_x x^2 + b_x x + a_y y^2 + b_y y + c \geq 0$ wird transformiert in

$$(\exists m,n: m = 2a_x x + b_x \wedge n = 2a_x y + b_y \wedge a_y m^2 + a_x n^2 \geq a_y b_x^2 + a_x b_y^2 - 4a_x a_y c).$$

Dabei sind $a_x > 0$, $a_y > 0$, b_x, b_y und c Zahlen und x und y Variablen.

5.3.2 Affinisierung

Eine hyperbolische Ungleichung der Form $xy \geq c$ mit positiver Konstante c wird transformiert in

$$x \geq 1 \wedge y \geq 1 \wedge$$

$$(\forall i: 1 \leq i \leq \lceil \sqrt{c} - 1 \rceil : line\left(\left\langle i, \left\lceil \frac{c}{i} \right\rceil \right\rangle, \left\langle i+1, \left\lceil \frac{c}{i+1} \right\rceil \right\rangle\right) \geq 0 \wedge line\left(\left\langle \left\lceil \frac{c}{i} \right\rceil, i \right\rangle, \left\langle \left\lceil \frac{c}{i+1} \right\rceil, i+1 \right\rangle\right) \leq 0) \vee$$

$$x \leq -1 \wedge y \leq -1 \wedge$$

$$(\forall i: \lfloor 1 - \sqrt{c} \rfloor \leq i \leq -1 : line\left(\left\langle i, \left\lceil \frac{c}{i} \right\rceil \right\rangle, \left\langle i-1, \left\lceil \frac{c}{-1} \right\rceil \right\rangle\right) \geq 0 \wedge line\left(\left\langle \left\lceil \frac{c}{i} \right\rceil, i \right\rangle, \left\langle \left\lceil \frac{c}{i-1} \right\rceil, i-1 \right\rangle\right) \leq 0)$$

Dabei ist $line(\langle x_1, y_1 \rangle, \langle x_2, y_2 \rangle) := (x_2 - x_1)(y - y_1) - (y_2 - y_1)(x - x_1)$.

Das ist ein Spezialfall des Affinierungssatzes (affinisation theorem [MP94]). Eine hyperbolische Ungleichung der Form $xy \geq c$ mit nichtpositiver Konstante c wird transformiert in

$\neg(x'y' \geq c')$, wobei $x' = -x$, $y' = y$ und $c' = 1 - c \geq 1$.

Die Affinisierung elliptischer Ungleichungen ist ähnlich der Affinisierung hyperbolischer Ungleichungen.

Eine hyperbolische Gleichung der Form $xy = c$ wird transformiert in

$$(\forall t: 1 \leq t \leq \lfloor \sqrt{|c|} \rfloor \wedge t | c: (x = t \wedge y = \frac{c}{t}) \vee (x = \frac{c}{t} \wedge y = t) \vee (x = -t \wedge y = -\frac{c}{t}) \vee (x = -\frac{c}{t} \wedge y = -t)).$$

Elliptische Gleichungen werden ähnlich behandelt.

Durch die Affinisierung werden polynomiale constraints in lineare constraints auf Kosten der Anzahl der constraints transformiert. Die Anzahl der generierten constraints ist von der Größenordnung $O(\sqrt{c})$.

5.4 Uninterpretierte Funktionssymbole

Arrayzugriffe führen zu Ausdrücken mit uninterpretierten Funktionssymbolen. Die Arrayvariablen werden als uninterpretierte Funktionssymbole bezeichnet, weil sie Funktionen sind, über die man nichts sonst weiß, d.h. nur $i = j \Rightarrow b(i) = b(j)$. Presburger-Arithmetik mit uninterpretierten Funktionssymbolen ist nicht entscheidbar [D72]. Man muß sich deshalb auf eine eingeschränkte Klasse von constraints mit uninterpretierten Funktionssymbolen beschränken [PW95]. Eine Einschränkung ist die Linearität der Argumente bzw. Indexausdrükke. Durch folgenden Algorithmus werden uninterpretierte Funktionssymbole, die die Voraussetzungen der Schritte 1-3 erfüllen, eliminiert. (Uninterpretierte Funktionssymbole, die keine der Voraussetzungen der Schritte 1-3 erfüllen, können nicht eliminiert werden.)

1. falls ein constraint nur ein uninterpretiertes Funktionssymbol enthält, wird es durch eine neue, skalare Variable ersetzt.
   Beispiel: $m > b(j) \wedge m \geq 1$ wird ersetzt durch $m > b_j \wedge m \geq 1$
2. falls ein constraint mehrere uninterpretierte Funktionssymbole enthält, deren Indexausdrücke gleich sind, so werden sie durch eine neue, skalare Variable ersetzt. Ob die Indexausdrücke gleich sind, ist nicht entscheidbar. Zwar sind die Indexausdrücke linear, aber die darin enthaltenen Variablen können in einer Relation stehen, die es nicht erlaubt, zu entscheiden, ob die Indexausdrücke unter allen Belegungen der Variablen gleich sind.
   Beispiel:
   $i - 1 = j + 1 \wedge b(i) < m \wedge b(j + 2) \geq 1$ wird ersetzt durch $i - 1 = j + 1 \wedge b_i < m \wedge b_i \geq 1$
3. falls ein constraint mehrere uninterpretierte Funktionssymbole enthält, deren Indexausdrücke verscheiden sind, so wird jedes uninterpretierte Funktionssymbol durch eine neue, skalare Variable ersetzt. Hier gilt das gleiche wie im vorigen Punkt.
   Beispiel: $i < j - 1 \wedge b(i - 1) < m \wedge b(j - 1) \geq 1$ wird ersetzt durch
   $$i < j - 1 \wedge b_i < m \wedge b_j \geq 1$$
4. Falls noch uninterpretierte Funktionssymbole übrig bleiben, terminiert das Verfahren ohne Erfolg.

5.5 Eliminierung innerer Quantoren

Um das Problem auf eine Form zu bringen, die Einschränkung 1 genügt, müssen innere Quantoren eliminiert werden. [PW93] betrachten constraints mit maximal 2 geschachtelten alternierenden Quantoren.
Zu zeigen ist die FC $(\exists x: L_i \wedge \cdots \wedge L_n \wedge \neg L)$ (siehe 0) bzw. $(\exists x: L_i \wedge \cdots \wedge L_n)$, wobei die L_i Literale sind. Falls die L_i keinen Quantor enthalten, gibt es nichts zu eliminieren.
Falls ein L_i einen Existenzquantor enthält, kann dieser eliminiert werden und die quantifizierten Variablen unter den äußersten Existenzquantor geschoben werden. Das ist eine gültige Transformation, da man durch gebundene Umbenennung die quantifizierten variablen von L_i verschieden von allen anderen Variablen machen kann.
Falls *L* einen Allquantor enthält, kann dieser durch Reinziehen der Negation in einen Existenzquantor transformiert werden.

Falls ein L_i einen Allquantor enthält, kann dieser durch Rausziehen der Negation in einen Existenzquantor transformiert werden. Nach diesen Transformationen hat man eine Formel der folgenden Form

$$(\exists x: L_1 \wedge \cdots L_i \wedge \neg(\exists y: L_{i+1}) \wedge \cdots \neg(\exists z: L_n))$$

Dabei sind die L_i Literale ohne Quantor. D.h. einige Literale sind mit einem Existenzquantor versehen und negiert. Zur Eliminierung der inneren Existenzquantoren aus einer Formel dieser Form wird das Verfahren aus [PW93] verwendet. Die detaillierte Beschreibung des Verfahrens geht über den Rahmen dieser Arbeit hinaus, doch die Grundidee wird kurz dargestellt. Existenzquantifizierte Variablen werden durch Verwendung des *floor*- und *ceiling*-Operators eliminiert. Die Negation kann daraufhin eliminiert werden. Allerdings ist dadurch Einschränkung 5 durch die *floor*- und *ceiling*-Operatoren verletzt. Diese werden wieder durch Einführung existenzquantifizierter Variablen eliminiert. Das Verfahren funktioniert für Literale der Form $x = na$ und $a \le nx \wedge mx \le b$. Es gilt $(\exists a : x = na) \equiv \left\lceil \frac{x}{n} \right\rceil \le \left\lfloor \frac{x}{n} \right\rfloor$ und $(\exists x : a \le nx \wedge mx \le b) \equiv \left\lceil \frac{a}{n} \right\rceil \le \left\lfloor \frac{b}{m} \right\rfloor$ (falls $a = 1 \vee b = 1$, gilt

$\left\lceil \frac{a}{n} \right\rceil \le \left\lceil \frac{b}{m} \right\rceil \equiv am \le bn$, was in diesem Fall vorzuziehen ist). Constraints mit *floor*- und *ceiling*-Operatoren heißen *quasi-linear*. Die Negation eines quasi-linearen constraints ist ein quasi-linearer constraint, z.B. ist $\neg(\exists a : x = na) \equiv \left\lceil \frac{x}{n} \right\rceil > \left\lfloor \frac{x}{n} \right\rfloor$. *floor*- und *ceiling*-Operationen werden durch Einführung neuer existenzquantifizierter Variablen, sogenannter *wildcard-Variablen*, eliminiert. Es gilt $E(\left\lceil \frac{a}{b} \right\rceil) \equiv (\exists c : E(c) \wedge bc - b < a \le bc)$ und $E(\left\lfloor \frac{a}{b} \right\rfloor) \equiv (\exists f : bf \le a < bf + b)$. Damit ergibt sich $\neg(\exists a : x = na) \equiv (\exists c, f : c > f \wedge nc - n < x \le nc \wedge nf \le x < nf + n)$. Im Beispiel 5 in Abschnitt 5.8 wird dieses Verfahren innerhalb eines etwas größeren Beispiels noch einmal illustriert.

5.6 Eliminierung von $\vee$

Nach der Eliminierung der inneren Quantoren kann es sein, daß die Formel eine Disjunktion enthält. Der Ausdruck wird dann in DNF transformiert und hat folgende Form:

$$(\exists x : K_{11} \wedge \ldots \wedge K_{1k1} \vee \cdots \vee K_{d1} \wedge \ldots \wedge K_{dkd})$$

Dieser wird in eine Disjunktion von lipps transformiert:

$$(\exists x: K_{11} \wedge \ldots \wedge K_{1k1}) \vee \cdots \vee (\exists x: K_{d1} \wedge \ldots \wedge K_{dkd})$$

Das ursprüngliche lipp ist lösbar, gdw. eines der erzeugten lipps lösbar ist.

5.7 Lokalisierung einer verletzten Zusicherung

Eine verletzte Zusicherung wird durch Ausführung des Programms aus einem Anfangszustand lokalisiert. Ein Anfangszustand ist eine Belegung der Programm- und Spezifikationsvariablen vor Ausführung des Programms, die die Vorbedingung *Pre* erfüllt.
Aus einer Lösung $Sol(ipp)$ des ipp soll ein Anfangszustand ermittelt werden, der zu einer verletzten Zusicherung führt. Das ist nur möglich, wenn $Sol(ipp)$ in Idealform vorliegt. In diesem Fall enthält $Sol(ipp)$ eine Variablenbelegung. Diese kann im Allgemeinen nicht als Anfangszustand verwendet werden, da sowohl *Pre* Variablen enthalten kann, die nicht in $Sol(ipp)$ sind und umgekehrt (siehe Beispiele). Daher müssen die Variablen in der Menge $Add := free(Sol(ipp)) \setminus free(Pre)$ eliminiert und die Variablen in der Menge $free(Pre) \setminus free(Sol(ipp))$ mit einem Wert belegt werden. Dazu werden die Werte der Variablen aus $Sol(ipp)$ in *Pre* eingesetzt und das so entstandene ipp wird erneut gelöst. Dieses ist aber im Allgemeinen einfacher als das ursprüngliche ipp oder sogar trivial in dem Sinne, daß es schon eine Variablenbelegung darstellt. Anschließend werden die Konjunkte mit Variablen aus *Add* entfernt. Formal sieht das Verfahren so aus:

Es wird eine Funktion *BM* definiert, die einer Variablenbelegung eine Menge von Paaren aus Variable und ihrem Wert zuordnet:

$$BM(\bigwedge_{i=1}^{n} v_i = w_i) = \{(v_i, w_i), i = 1,\ldots,n\}.$$

Ein Anfangszustand ist dann

$$BM^{-1}((BM(Sol(ipp)) \cup BM(Sol(Pre_{w_i}^{v_i}))) \setminus \{(v_i, w_i), v_i \in Add\}).$$

Die folgenden Beispiele erläutern die Ermittlung eines Anfangszustandes, der zu einer verletzten Zusicherung führt.

5.8 Beispiele

Beispiel 1: Lineare constraints mit einem äußeren Existenzquantor

```
subtype t is integer range 0..100;
x, y: t;
-- 0 ≤ x ≤ 100 ∧ 0 ≤ y ≤ 100 ∧ x = a ∧ y = b
x := x + y;
y := x - y;
x := x - y;
-- 0 ≤ x ≤ 100 ∧ 0 ≤ y ≤ 100 ∧ x = b ∧ y = a
```

Die VC lautet

$$[0 \le x \le 100 \wedge 0 \le y \le 100 \wedge x = a \wedge y = b \Rightarrow$$
$$0 \le y \le 100 \wedge 0 \le x + y - y \le 100 \wedge y = b \wedge x = a \wedge 0 \le x + y \le 100]$$

Die nicht beweisbare Klausel lautet

$$[0 \le x \le 100 \wedge 0 \le y \le 100 \wedge x = a \wedge y = b \Rightarrow x + y \le 100]$$

Das zugehörige lipp lautet

$$(\exists x, y, a, b: 0 \le x \le 100 \wedge 0 \le y \le 100 \wedge x = a \wedge y = b \wedge x + y \ge 101)$$

Eine Lösung $Sol(ipp)$ ist $x = 1 \wedge a = 1 \wedge y = 100 \wedge b = 100$. Ein Anfangszustand ist

$BM^{-1}((BM(Sol(ipp)) \cup BM(Sol(Pre_{w_i}^{v_i}))) \setminus \{(v_i, w_i), v_i \in Add\})$

=

$BM^{-1}((BM(x=1 \wedge a=1 \wedge y=100 \wedge b=100) \cup BM(Sol(true))) \setminus \{(v_i, w_i), v_i \in \varnothing\})$

=

$x=1 \wedge a=1 \wedge y=100 \wedge b=100$

Damit ist gezeigt, daß das Programm nicht korrekt ist.

Beispiel 2: Nichtlineare constraints

```
-- x ≥ 0 ∧ y > 0 ∧ q = 0 ∧ r = x
while r >= y loop
   -- Inv: 0 ≤ r ∧ y > 0 ∧ qy + r = x; term: r+1
   r := r - y; q := q + r;
end loop;
-- 0 ≤ r < y ∧ qy + r = x
```

1. $[V \Rightarrow I] \equiv true$
2. $[\neg B \wedge I \Rightarrow N] \equiv true$
3. $[B \wedge I \Rightarrow wp(S, I)]$

$[r \geq y \wedge 0 \leq r \wedge y > 0 \wedge qy + r = x \Rightarrow 0 \leq r - y \wedge y > 0 \wedge (q + r - y)y + r - y = x]$

≡

$[r \geq y > 0 \wedge qy + r = x \Rightarrow 0 < y \leq r \wedge qy + ry - y^2 + r - y = x]$

≡

$[(r \geq y > 0 \wedge qy + r = x \Rightarrow 0 < y \leq r) \wedge$

$(r \geq y > 0 \wedge qy + r = x \Rightarrow qy + ry - y^2 + r - y = x)]$

≡

$[r \geq y > 0 \wedge qy + r = x \Rightarrow qy + ry - y^2 + r - y = x]$

≡

$[r \geq y > 0 \Rightarrow qy + ry - y^2 + r - y = qy + r]$

≡

$[r \geq y > 0 \Rightarrow ry - y^2 - y = 0]$

Diese nicht beweisbare Klausel führt zu folgendem lipp:

$(\exists r, y: r \geq y \wedge y > 0 \wedge ry - y^2 - y \neq 0)$

≡ Abschnitt 5.2

$(\exists r, y: r \geq y \wedge y \geq 1 \wedge (-1 \geq ry - y^2 - y \vee ry - y^2 - y \geq 1))$

≡ Abschnitt 5.6

$(\exists r, y: r \geq y \wedge y \geq 1 \wedge -1 \geq ry - y^2 - y) \vee (\exists r, y: r \geq y \wedge y \geq 1 \wedge ry - y^2 - y \geq 1)$

Es genügt, die Lösbarkeit eines Disjunkts zu zeigen, o.B.d.A. des ersten. Nur Einschränkung 3 ist nicht erfüllt, es muß also

$(\exists r, y: r \geq y \wedge y \geq 1 \wedge -1 \geq ry - y^2 - y)$

affinisiert werden (Abschnitt 5.3) . Faktorisierung (Abschnitt 5.3.1) von

$-1 \geq ry - y^2 - y \equiv -ry + y^2 + y \geq 1$ ergibt $(\exists z: z = 1 + y - r \wedge zy \geq 1)$.

Die darin enthaltene hyperbolische Ungleichung mit positiver Konstante wird mit Abschnitt 5.3.2 transformiert in

$z \geq 1 \wedge y \geq 1 \vee z \leq -1 \wedge y \leq -1.$

Damit ist der constraint linearisiert und das 1. Disjunkt wird zu
$(\exists r, y: r \geq y \wedge y \geq 1 \wedge (\exists z: z = 1 + y - r \wedge (z \geq 1 \wedge y \geq 1 \vee z \leq -1 \wedge y \leq -1))$

≡ Abschnitt 5.5

$(\exists r, y, z: r \geq y \wedge y \geq 1 \wedge z = 1 + y - r \wedge z \geq 1 \wedge y \geq 1 \vee r \geq y \wedge y \geq 1 \wedge z = 1 + y - r \wedge z \leq -1 \wedge y \leq -1))$
≡ Abschnitt 5.6

$(\exists r, y, z: r \geq y \wedge y \geq 1 \wedge z = 1 + y - r \wedge z \geq 1 \wedge y \geq 1) \vee$
$(\exists r, y, z: r \geq y \wedge y \geq 1 \wedge z = 1 + y - r \wedge z \leq -1 \wedge y \leq -1))$

Es genügt, eines der lipps zu lösen. Eine Lösung $Sol(ipp)$ des 1. lipp ist $z = y = r = 1$. (Das 2. lipp hat keine Lösung.) Ein Anfangszustand ist

$$BM^{-1}((BM(Sol(ipp)) \cup BM(Sol(Pre_{w_i}^{v_i}))) \setminus \{(v_i, w_i), v_i \in Add\})$$

=

$$BM^{-1}((\{(z,1),(y,1),(r,1) \cup BM(Sol(x \geq 0 \wedge x = 1 \wedge q = 0))) \setminus \{(z,1))$$

=

$$x = 1 \wedge y = 1 \wedge r = 1 \wedge q = 0$$

Da der Anfangszustand aus einer approximierten VC gewonnen wurde, muß man wegen Lemma 5.4 das Programm damit noch ausführen. Die Nachbedingung wird verletzt. Damit ist gezeigt, daß das Programm nicht korrekt ist. (Dieses gilt auch für die folgenden Beispiele und wird nicht mehr erwähnt.)

Beispiel 3: Nichtlineare constraints

```
-- z = 0 ∧ 0 ≤ a ≤ 10 ∧ -10 ≤ b ≤ 10
for i in 1 .. a loop
  z := z + b;
  -- z = ib ∧ 0 ≤ a ≤ 10 ∧ -10 ≤ b ≤ 10
end loop;
-- z = ab ∧ -100 ≤ z ≤ 10
```

Die Klausel, die nicht bewiesen werden kann, lautet

$$[0 \leq a \leq 10 \wedge -10 \leq b \leq 10 \Rightarrow ab \leq 10].$$

Das führt zu dem ipp

$$(\exists a, b: 0 \leq a \leq 10 \wedge -10 \leq b \leq 10 \wedge ab \geq 11)$$

mit einer hyperbolischen Ungleichung mit positiver Konstante. Abschnitt 5.3.2 ergibt für $ab \geq 11$:

$a \geq 1 \wedge b \geq 1 \wedge line(<1,11>,<2,6>) \geq 0 \wedge line(<11,1>,<6,2>) \leq 0 \wedge$
$line(<2,6>,<3,4>) \geq 0 \wedge line(<6,2>,<4,3>) \leq 0 \wedge$
$line(<3,4>,<4,3>) \geq 0 \wedge line(<4,3>,<3,4>) \leq 0 \vee$
$a \leq -1 \wedge b \leq -1 \wedge line(<-3,-4>,<-4,-3>) \geq 0 \wedge line(<-4,-3>,<-3,-4>) \leq 0 \wedge$
$line(<-2,-6>,<-3,-4>) \geq 0 \wedge line(<-6,-2>,<-4,-3>) \leq 0 \wedge$
$line(<-1,-11>,<-2,-6>) \geq 0 \wedge line(<-11,-1>,<-6,-2>) \leq 0$

≡ Definition von *line*

$$a \geq 1 \wedge b \geq 1 \wedge (2-1)(b-11)-(6-11)(a-1) \geq 0 \wedge (6-11)(b-1)-(2-1)(a-11) \leq 0 \wedge$$
$$(3-2)(b-6)-(4-6)(a-2) \geq 0 \wedge (4-6)(b-2)-(3-2)(a-6) \leq 0 \wedge$$
$$(4-3)(b-4)-(3-4)(a-3) \geq 0 \wedge (3-4)(b-3)-(4-3)(a-4) \leq 0 \vee$$
$$a \leq -1 \wedge b \leq -1 \wedge (-4+3)(b+4)-(-3+4)(a+3) \geq 0 \wedge (-3+4)(b+3)-(-4+3)(a+4) \leq 0 \wedge$$
$$(-3+2)(b+6)-(-4+6)(a+2) \geq 0 \wedge (-4+6)(b+2)-(-3+2)(a+6) \leq 0 \wedge$$
$$(-2+1)(b+11)-(-6+11)(a+1) \geq 0 \wedge (-6+11)(b+1)-(-2+1)(a+11) \leq 0$$

≡ Arithmetik

$$a \geq 1 \wedge b \geq 1 \wedge b+5a \geq 16 \wedge a+5b \geq 16 \wedge b+2a \geq 10 \wedge a+2b \geq 10 \wedge a+b \geq 7 \vee$$
$$a \leq -1 \wedge b \leq -1 \wedge -b-5a \geq 16 \wedge -a-5b \geq 16 \wedge -b-2a \geq 10 \wedge -a-2b \geq 10 \wedge -a-b \geq 7$$

Damit ergibt sich für $(\exists a,b{:}0 \leq a \leq 10 \wedge -10 \leq b \leq 10 \wedge ab \geq 11)$ in DNF

$$(\exists a,b : a \geq 0 \wedge 10 \geq a \wedge b \geq -10 \wedge 10 \geq b \wedge$$
$$a \geq 1 \wedge b \geq 1 \wedge b+5a \geq 16 \wedge a+5b \geq 16 \wedge b+2a \geq 10 \wedge a+2b \geq 10 \wedge a+b \geq 7 \vee$$
$$a \geq 0 \wedge 10 \geq a \wedge b \geq -10 \wedge 10 \geq b \wedge$$
$$a \leq -1 \wedge b \leq -1 \wedge -b-5a \geq 16 \wedge -a-5b \geq 16 \wedge -b-2a \geq 10 \wedge -a-2b \geq 10 \wedge -a-b \geq 7)$$

≡ Abschnitt 5.6

$$(\exists a,b : a \geq 0 \wedge 10 \geq a \wedge b \geq -10 \wedge 10 \geq b \wedge$$
$$a \geq 1 \wedge b \geq 1 \wedge b+5a \geq 16 \wedge a+5b \geq 16 \wedge b+2a \geq 10 \wedge a+2b \geq 10 \wedge a+b \geq 7) \vee$$
$$(\exists a,b : a \geq 0 \wedge 10 \geq a \wedge b \geq -10 \wedge 10 \geq b \wedge$$
$$a \leq -1 \wedge b \leq -1 \wedge -b-5a \geq 16 \wedge -a-5b \geq 16 \wedge -b-2a \geq 10 \wedge -a-2b \geq 10 \wedge -a-b \geq 7)$$

Damit ist das ursprüngliche Problem auf 2 lipps transformiert worden. Es genügt, eines davon zu lösen, o.B.d.A. das erste. Eine Lösung ist $a = 10 \wedge b = 10$. Ein zu einem Fehler führender Anfangszustand ist dann $a = 10 \wedge b = 10 \wedge z = 0$.

Beispiel 4: Division (linearisierbare Nichtlinearität)

```
-- b ≥ 0
if a/100 >= b then a := a - 100*b;
else a := a + 100*b;
end if;
-- a ≥ 0
```

Die nicht beweisbare VC lautet

$$[b \geq 0 \Rightarrow \frac{a}{100} \geq b \wedge a \geq 100b \vee \frac{a}{100} < b \wedge a + 100b \geq 0].$$

Die nicht beweisbare Klausel lautet

$$[b \geq 0 \wedge \frac{a}{100} < b \Rightarrow a + 100b \geq 0].$$

Das ipp lautet

$$(\exists a,b{:}b \geq 0 \wedge \frac{a}{100} < b \wedge a + 100b < 0)$$

≡ Abschnitt 5.2

$$(\exists a,b{:}b \geq 0 \wedge (\exists t,r{:}100t + r = a \wedge r \geq 0 \wedge 99 \geq r \wedge b - 1 \geq t) \wedge -1 \geq a + 100b)$$

≡ Abschnitt 5.5

$$(\exists a,b,t,r: b \geq 0 \wedge 100t + r = a \wedge r \geq 0 \wedge 99 \geq r \wedge b - 1 \geq t \wedge -1 \geq a + 100b)$$

Damit hat man ein lipp. Eine Lösung ist

$$a = -200 \wedge b = 1 \wedge t = -2 \wedge r = 0.$$

Ein zu einem Fehler führender Anfangszustand ist dann $a = -200 \wedge b = 1$.

Beispiel 5: Innere Quantoren

```
-- b ≥ 0 ∧ x = a ∧ y = b ∧ z = 0
while y>0 loop
   -- Inv: y ≥ 0 ∧ z + xy = ab; term: y
   if even(y) then y := y/2; x := 2*x;
   else y := y-2; z := z + x;
   end if;
end loop;
-- z = a*b
```

Der nicht beweisbare Teil des Korrektheitsbeweises lautet

$$[y > 0 \wedge z + xy = ab \Rightarrow even(y) \wedge \frac{y}{2} \geq 0 \wedge z + 2x\frac{y}{2} = ab \vee$$
$$odd(y) \wedge y - 2 \geq 0 \wedge z + x + x(y-2) = ab]$$

Nach der Zerlegung dieser Formeln in Klauseln bleibt als nicht beweisbare Klausel

$$[y > 0 \wedge z + xy = ab \wedge odd(y) \Rightarrow y - 2 \geq 0].$$

Der Beweiser wendet solange Transformationsregeln an, bis die Klausel entweder *true* ist oder sich keine weitere Regel mehr anwenden läßt. In diesem Fall kann der Beweiser den Term mit *z* nach *z* auflösen und *z* substituieren. Das ergibt

$$[y > 0 \wedge odd(y) \Rightarrow y - 2 \geq 0].$$

Als ipp erhält man

$$(\exists y: y > 0 \wedge odd(y) \wedge y - 2 < 0)$$

$\equiv$ Abschnitt 5.2

$$(\exists y: y \geq 1 \wedge (\exists i: y = 2i + 1) \wedge -1 \geq y - 2)$$

$\equiv$ Abschnitt 5.5

$$(\exists y,i: y \geq 1 \wedge y = 2i + 1 \wedge -1 \geq y - 2)$$

Jetzt liegt ein lipp vor, das mit Hilfe des **Ω**-Tests gelöst werden kann. Als eine Lösung erhält man $y = 1 \wedge i = 0$. Eine Vorbedingung, die zu einer verletzten Invariante am Ende des Schleifenrumpfes führt, ist $x = y = a = b = 1 \wedge z = 0$.

•

```
-- n ≥ 2 ∧ p
for i in 3 .. n-1 loop
   if n mod i = 0 then p := false; end if;
   -- p = (∀j: 2≤j≤i: n mod j ≠ 0)
end loop;
-- p = prim(n)
```

Die nicht beweisbare VC lautet

$$[n \geq 2 \wedge p \wedge 3 \leq n - 1 \Rightarrow n \bmod 3 = 0 \wedge \neg(\forall j: 2 \leq j \leq 3: n \bmod j \neq 0) \vee$$
$$n \bmod 3 \neq 0 \wedge p = (\forall j: 2 \leq j \leq 3: n \bmod j \neq 0)]$$

Die nicht beweisbare Klausel lautet

$$[n \geq 2 \wedge 3 \leq n-1 \wedge n \bmod 3 \neq 0 \Rightarrow n \bmod 2 \neq 0]\,.$$

Das ergibt das ipp

$$(\exists n{:}\, n \geq 2 \wedge 3 \leq n-1 \wedge n \bmod 3 \neq 0 \wedge n \bmod 2 = 0)$$

$\equiv$ Abschnitt 5.2

$$(\exists n{:}\, n \geq 4 \wedge \neg(\exists j{:}\, 3j = n) \wedge (\exists j{:}\, 2j = n))$$

$\equiv$ Abschnitt 5.5

$$(\exists n, j{:}\, n \geq 4 \wedge \neg(\exists j{:}\, n \leq 3i \leq n) \wedge 2j = n)$$

$\equiv$ Abschnitt 5.5

$$(\exists n, i{:}\, n \geq 4 \wedge \neg(\left\lceil \frac{n}{3} \right\rceil \leq \left\lfloor \frac{n}{3} \right\rfloor) \wedge 2j = n)$$

$\equiv$ Abschnitt 5.5

$$(\exists n, j{:}\, n \geq 4 \wedge \left\lceil \frac{n}{3} \right\rceil > \left\lfloor \frac{n}{3} \right\rfloor \wedge 2j = n)$$

$\equiv$ Abschnitt 5.5

$$(\exists n, j{:}\, n \geq 4 \wedge (\exists c, f{:}\, 3c - c < n \leq 3c \wedge c > f \wedge 3f \leq n < 3f + 3 \wedge 2j = n))$$

$\equiv$ Abschnitt 5.5

$$(\exists n, j, c, f{:}\, n \geq 4 \wedge 3c - c < n \leq 3c \wedge c > f \wedge 3f \leq n < 3f + 3 \wedge 2j = n)$$

Damit hat man das ipp auf ein lipp reduziert und kann mit Hilfe des Ω-Tests eine Lösung finden, z.B. $i = 2 \wedge n = 4 \wedge c = 1 \wedge f = 2$. Ein Anfangszustand, der zu einem falschen Ergebnis oder undefinierten Zwischenzustand führt, ist $n = 4 \wedge p \equiv true$.

Beispiel 6: Uninterpretierte Funktionssymbole und innere Quantoren

```
-- m = b(1)
for i in 1 .. n loop
  if m > b(i) then m := b(i); end if;
  -- (∀j: 1≤j≤i: m ≥ b(j)) ∧ (∃j: 1≤j≤i: m = b(j))
end loop;
-- (∀j: 1≤j≤n: m ≥ b(j)) ∧ (∃j: 1≤j≤n: m = b(j))
```

Der nicht beweisbare Teil des Korrektheitsbeweises lautet

$$[2 \leq i \leq n \wedge (\forall j : 1 \leq j \leq i-1 : m \geq b(j)) \wedge (\exists j : 1 \leq j \leq i-1 : m = b(j)) \Rightarrow$$
$$m > b(i) \wedge (\forall j : 1 \leq j \leq i : b(i) \geq b(j)) \vee$$
$$m \leq b(i) \wedge (\forall j : 1 \leq j \leq i : m \geq b(j)) \wedge (\exists j : 1 \leq j \leq i : m = b(j))]$$

Von den Klauseln, in die diese Formel transformiert wird, ist eine der nicht-beweisbaren

$$[2 \leq i \leq n \wedge (\forall j : 1 \leq j \leq i-1 : m \geq b(j)) \wedge (\exists j : 1 \leq j \leq i-1 : m = b(j)) \wedge m > b(i) \Rightarrow$$
$$(\forall j : 1 \leq j \leq i : b(i) \geq b(j))]$$

$\equiv$ Skolemisierung

$$[2 \leq i \leq n \wedge (\forall j : 1 \leq j \leq i-1 : m \geq b(j)) \wedge 1 \leq j' \leq i-1 \wedge m = b(j') \wedge m > b(i) \wedge 1 \leq j'' \leq i \Rightarrow$$
$$b(i) \geq b(j'')]$$

≡ Substitution

$$[2 \le i \le n \wedge (\forall j : 1 \le j \le i-1 : b(j') \ge b(j)) \wedge 1 \le j' \le i-1 \wedge b(j') > b(i) \wedge 1 \le j'' \le i \Rightarrow b(i) \ge b(j'')]$$

Zunächst wird $(\exists j{:}1 \le j \le i-1{:}\, b1 < b2 \wedge j \ne j')$ quantorfrei gemacht.

$$(\exists j{:}1 \le j \le i-1{:}\, b1 < b2 \wedge j \ne j')$$

≡ Logik

$$(\exists j{:}1 \le j \le i-1 \wedge j \ne j') \wedge b1 < b2$$

≡ Logik

$$(\exists j{:}1 \le j \le i-1 \wedge (j < j' \vee j > j')) \wedge b1 < b2$$

≡ Logik

$$(\exists j{:}1 \le j \le i-1 \wedge j < j' \vee 1 \le j \le i-1 \wedge j > j') \wedge b1 < b2$$

≡ Logik

$$(\exists j{:}\, i \le j' \wedge 1 \le j \le i-1 \vee i \ge j'+1 \wedge 1 \le j \le j'-1 \vee j' \ge 0 \wedge j'+1 \le j \le i-1 \vee j' \le -1 \wedge 1 \le j \le i-1) \wedge b1 < b2$$

≡ Logik

$$(i \le j' \wedge (\exists j{:}1 \le j \le i-1) \vee i \ge j'+1 \wedge (\exists j{:}1 \le j \le j'-1) \vee j' \ge 0 \wedge (\exists j{:}\, j'+1 \le j \le i-1) \vee j' \le -1 \wedge (\exists j{:}1 \le j \le i-1)) \wedge b1 < b2$$

≡ Abschnitt 5.5

$$(i \le j' \wedge 1 \le i-1 \vee i \ge j'+1 \wedge 1 \le j'-1 \vee j' \ge 0 \wedge j'+1 \le i-1 \vee j' \le -1 \wedge 1 \le i-1) \wedge b1 < b2$$

Das zugehörige ipp lautet

$$(\exists i, n, j', j'' {:}\, 2 \le i \le n \wedge (\forall j{:}1 \le j \le i-1{:}\, b(j') \ge b(j)) \wedge 1 \le j' \le i-1 \wedge b(j') > b(i) \wedge 1 \le j'' \le i \wedge b(i) < b(j''))$$

≡

$$(\exists i, n, j', j'' {:}\, 2 \le i \le n \wedge (\forall j{:}1 \le j \le i-1{:}\, (j = j' \vee j \ne j') \wedge b(j') \ge b(j)) \wedge 1 \le j' \le i-1 \wedge b(j') > b(i) \wedge 1 \le j'' \wedge (j'' < i \vee j'' = i) \wedge b(i) < b(j''))$$

≡ Logik, Abschnitt 5.6

$$(\exists i, n, j', j'' {:}\, 2 \le i \le n \wedge (\forall j{:}1 \le j \le i-1{:}\, j \ne j' \wedge b(j') \ge b(j) \vee j = j' \wedge b(j') \ge b(j)) \wedge 1 \le j' \le i-1 \wedge b(j') > b(i) \wedge 1 \le j'' \wedge j'' < i \wedge b(i) < b(j'')) \vee$$
$$(\exists i, n, j', j'' {:}\, 2 \le i \le n \wedge (\forall j{:}1 \le j \le i-1{:}\, j \ne j' \wedge b(j') \ge b(j) \vee j = j' \wedge b(j') \ge b(j)) \wedge 1 \le j' \le i-1 \wedge b(j') > b(i) \wedge 1 \le j'' \wedge j'' = i \wedge b(i) < b(j''))$$

≡

$$(\exists i,n,j',j'':2 \le i \le n \wedge (\forall j:1 \le j \le i-1:b(j') \ge b(j) \vee j = j') \wedge$$
$$1 \le j' \le i-1 \wedge b(j') > b(i) \wedge 1 \le j'' \wedge j'' < i \wedge b(i) < b(j'')) \vee$$
$$(\exists i,n,j',j'':2 \le i \le n \wedge (\forall j:1 \le j \le i-1:b(j') \ge b(j) \vee j = j') \wedge$$
$$1 \le j' \le i-1 \wedge b(j') > b(i) \wedge 1 \le j'' \wedge j'' = i \wedge b(i) < b(i))$$

≡ Logik

$$(\exists i,n,j',j'':2 \le i \le n \wedge (\forall j:1 \le j \le i-1:b(j') \ge b(j) \vee j = j') \wedge$$
$$1 \le j' \le i-1 \wedge b(j') > b(i) \wedge 1 \le j'' \wedge j'' < i \wedge b(i) < b(j'')) \vee$$
$$false$$

≡ Abschnitt 5.4 ($b1 = b(j') \wedge b2 = b(j) \wedge b3 = b(i) \wedge b4 = b(j'')$)

$$(\exists i,n,j',j'',b1,b2,b3,b4:2 \le i \le n \wedge (\forall j:1 \le j \le i-1:b1 \ge b2 \vee j = j') \wedge$$
$$1 \le j' \le i-1 \wedge b1 > b3 \wedge 1 \le j'' \wedge j'' < i \wedge b3 < b4)$$

≡ Logik

$$(\exists i,n,j',j'',b1,b2,b3,b4:2 \le i \le n \wedge \neg(\exists j:1 \le j \le i-1:b1 < b2 \wedge j \ne j') \wedge$$
$$1 \le j' \le i-1 \wedge b1 > b3 \wedge 1 \le j'' \wedge j'' < i \wedge b3 < b4)$$

Das wird negiert und wieder eingesetzt

$$i \le 1 \wedge j' \ge 0 \wedge b1 \ge b2 \vee j' = 0 \wedge i = 0 \wedge b1 \ge b2 \vee j' = 0 \wedge i = 1 \wedge b1 \ge b2 \vee j' = 1 \wedge i = 2 \wedge b1 \ge b2$$

Damit ist das Problem linearisiert und man erhält 4 lipps.

$$(\exists i,n,j',j'',b1,b2,b3,b4:2 \le i \le n \wedge i \le 1 \wedge j' \ge 0 \wedge b1 \ge b2 \wedge$$
$$1 \le j' \le i-1 \wedge b1 > b3 \wedge 1 \le j'' \wedge j'' < i \wedge b3 < b4) \vee$$
$$(\exists i,n,j',j'',b1,b2,b3,b4:2 \le i \le n \wedge j' = 0 \wedge i = 0 \wedge b1 \ge b2 \wedge$$
$$1 \le j' \le i-1 \wedge b1 > b3 \wedge 1 \le j'' \wedge j'' < i \wedge b3 < b4) \vee$$
$$(\exists i,n,j',j'',b1,b2,b3,b4:2 \le i \le n \wedge i \le 1 \wedge j' = 0 \wedge i = 1 \wedge b1 \ge b2 \wedge$$
$$1 \le j' \le i-1 \wedge b1 > b3 \wedge 1 \le j'' \wedge j'' < i \wedge b3 < b4) \vee$$
$$(\exists i,n,j',j'',b1,b2,b3,b4:2 \le i \le n \wedge j' = 1 \wedge i = 2 \wedge b1 \ge b2 \wedge$$
$$1 \le j' \le i-1 \wedge b1 > b3 \wedge 1 \le j'' \wedge j'' < i \wedge b3 < b4)$$

≡

$$false \vee false \vee false \vee$$
$$(\exists i,n,j',j'',b1,b2,b3,b4:2 \le i \le n \wedge j' = 1 \wedge i = 2 \wedge b1 \ge b2 \wedge$$
$$1 \le j' \le i-1 \wedge b1 > b3 \wedge 1 \le j'' \wedge j'' < i \wedge b3 < b4)$$

Eine Lösung ist

$$i = 2 \wedge n = 2 \wedge j' = 1 \wedge j'' = 1 \wedge b1 = 2 \wedge b2 = 1 \wedge b3 = 1 \wedge b4 = 2 .$$

Durch Rücksubstitution erhält man Werte für *b*:

$$i = 2 \wedge n = 2 \wedge j' = 1 \wedge j'' = 1 \wedge b(j') = 2 \wedge b(j) = 1 \wedge b(i) = 1 \wedge b(j'') = 2 .$$

Mit den Lösungswerten der Indizes ergibt sich (*j* und damit $b(j)$ existieren nicht mehr)

$$i = 2 \wedge n = 2 \wedge j' = 1 \wedge j'' = 1 \wedge b(1) = 2 \wedge b(2) = 1 .$$

Ein Anfangszustand, der zu einer nicht korrekten Nachbedingung oder einem undefinierten Zustand führt, ist

$$m = 2 \wedge n = 2 \wedge b(1) = 2 \wedge b(2) = 1.$$

Falls das Verfahren keine Variablenbelegung zurückliefert (*Versagen*), kann es das Problem soweit wie möglich lösen und den Rest ungelöst zurückgeben. Oft kann der menschliche Betrachter dann eine Lösung erkennen.

Beispiel 7: Versagen wegen nicht linearisierbarer Nichtlinearität

```
-- x = a ∧ y = b ∧ a ≥ 0 ∧ b ≥ 0 ∧ z = 1
while y>0 loop
   -- Inv: y ≥ 0 ∧ z * x^y = a^b; term: y
   if y mod 2 = 0 then y := y/2; x := x * x;
   else z := z * x;
   end if;
end loop;
-- z = a^b
```

Eine nicht beweisbare VC ist

$$[y > 0 \wedge zx^y = a^b \Rightarrow even(y) \wedge \frac{y}{2} \geq 0 \wedge z(x^2)^{y/2} = a^b \vee$$
$$odd(y) \wedge y \geq 0 \wedge zx^{y+1} = a^b]$$

Eine nicht beweisbare Klausel lautet

$$[y > 0 \wedge zx^y = a^b \wedge odd(y) \Rightarrow zx^{y+1} = a^b].$$

Das entsprechende ipp lautet

$$(\exists x, y, z, a, b: y > 0 \wedge zx^y = a^b \wedge (\exists i: y = 2i + 1) \wedge zx^{y+1} \neq a^b).$$

Es kann mit dem beschriebenen Verfahren nicht gelöst werden. Doch man kann „von Hand" weiterrechnen: einen Anfangszustand, der zu einer verletzten Zusicherung führt, erhält man durch Konjunktion des ipp mit der Vorbedingung.

$$(\exists x, y, z, a, b: x = a \wedge y = b \wedge z = 1 \wedge a \geq 0 \wedge b \geq 0 \wedge y > 0 \wedge zx^y = a^b \wedge (\exists i: y = 2i + 1) \wedge zx^{y+1} \neq a^b)$$
$$\equiv$$
$$(\exists x, y, z, a, b: x = a \wedge y = b \wedge z = 1 \wedge a \geq 0 \wedge b > 0 \wedge a^b = a^b \wedge (\exists i: y = 2i + 1) \wedge a^{b+1} \neq a^b)$$
$$\equiv$$
$$(\exists x, y, z, a, b, i: x = a \wedge y = b \wedge z = 1 \wedge a \geq 0 \wedge b > 0 \wedge b = 2i + 1 \wedge a^{b+1} \neq a^b)$$

Eine Lösung ist $x = a = 0 \wedge y = b = 1 \wedge z = 1$. Damit wird die Zusicherung nach dem Rumpf verletzt, die besagt, daß die Terminierungsfunktion nach Ausführung des Rumpfes einen kleineren Wert als vor Ausführung des Rumpfes haben muß.

•

```
-- x ≥ 0 ∧ y ≥ 0 ∧ x = a ∧ y = b
while x /= y loop
-- Inv: ggt(x, y) = ggt(a, b); term: x+y+1
   if x > y then x := x - y;
            else y := y - x;
   end if;
end loop;
-- x = ggt(a, b)
```

Die nicht beweisbare VC lautet

$$[x \neq y \wedge ggt(x, y) = ggt(a, b) \wedge x \geq 0 \wedge y \geq 0 \Rightarrow x > y \wedge y > 0 \vee x \leq y \wedge x > 0].$$

Eine nicht beweisbare Klausel (auch die einzige) lautet

$$[x < y \wedge ggt(x,y) = ggt(a,b) \wedge x \geq 0 \wedge y \geq 0 \Rightarrow x > 0].$$

Das führt zu dem ipp

$$(\exists x,y,a,b: y \geq x+1 \wedge ggt(x,y) = ggt(a,b) \wedge x \geq 0 \wedge y \geq 0 \wedge x \leq 0)$$

$$\equiv$$

$$(\exists x,y,a,b: y \geq x+1 \wedge ggt(x,y) = ggt(a,b) \wedge y \geq 0 \wedge x = 0)$$

$$\equiv$$

$$(\exists x,y,a,b: y \geq 1 \wedge ggt(0,y) = ggt(a,b) \wedge x = 0)$$

$$\equiv$$

$$(\exists x,y,a,b: y \geq 1 \wedge y = ggt(a,b) \wedge x = 0)$$

$\equiv$ Abschnitt 5.2

$$(\exists x,y,a,b: y \geq 1 \wedge y|a \wedge y|b \wedge (\forall t: t|a \wedge t|b \Rightarrow t \leq y) \wedge x = 0)$$

$\equiv$ Abschnitt 5.2

$$(\exists x,y,a,b: y \geq 1 \wedge (\exists i: yi = a) \wedge (\exists i: yi = b) \wedge (\forall t: (\exists i: ti = a) \wedge (\exists i: ti = b) \Rightarrow t \leq y) \wedge x = 0)$$

$$\equiv$$

$$(\exists x,y,a,b,k,l: y \geq 1 \wedge yk = a \wedge yl = b \wedge (\forall t,i,j: ti = a \wedge tj = b \Rightarrow t \leq y) \wedge x = 0)$$

$\equiv$ Abschnitt 5.5

$$(\exists x,y,a,b,k,l: y \geq 1 \wedge yk = a \wedge yl = b \wedge \neg(\exists t,i,j: ti = a \wedge tj = b \wedge t \geq y+1) \wedge x = 0)$$

Die nicht linearen Ausdrücke können nicht linearisiert werden. Eine Lösung „von Hand“ ist $x = 0 \wedge y = 1 \wedge k = a \wedge l = b$. Ein zu einer Fehlersituation führender Anfangszustand ist damit $x = 0 \wedge y = 1 \wedge a = 0 \wedge b = 1$. Er führt zum gleichen Fehler wie im vorigen Beispiel.

Beispiel 8: Versagen wegen zu komplexer Indexausdrücke

```
-- i = 1
while i <= 100 and b(100 / i) > b(i) loop
   i := i + 1;
   -- 1 ≤ i ≤ 101
end loop;
-- b(100 / i) < b(i) ∨ i = 101
```

Die nicht beweisbare VC lautet

$$[(i \geq 101 \vee b(\frac{100}{i}) \leq b(i)) \wedge 1 \leq i \leq 101 \Rightarrow b(\frac{100}{i}) < b(i) \vee i = 101].$$

Eine nicht beweisbare Klausel ist

$$[b(\frac{100}{i}) \leq b(i) \wedge 1 \leq i \leq 101 \wedge b(\frac{100}{i}) \geq b(i) \Rightarrow i = 101].$$

Sie führt zu dem ipp

$$(\exists i,b: b(\frac{100}{i}) \leq b(i) \wedge 1 \leq i \leq 101 \wedge b(\frac{100}{i}) \geq b(i) \wedge i \neq 101).$$

$$(\exists i,b: b(\frac{100}{i}) \leq b(i) \wedge 1 \leq i \leq 101 \wedge b(\frac{100}{i}) \geq b(i) \wedge i \neq 101)$$

≡ Logik

$$(\exists i,b{:}\, b(\frac{100}{i}) = b(i) \wedge 1 \le i \le 100)$$

≡ Abschnitt 5.2

$$(\exists i,b,t,a{:}\, b(t) = b(i) \wedge ti + a = 100 \wedge 0 \le a \le i - 1 \wedge 1 \le i \le 100)$$

≡ Abschnitt 5.4

$$(\exists i,b,t,a,s{:}\, t = i \wedge s = s \wedge ti + a = 100 \wedge 0 \le a \le i - 1 \wedge 1 \le i \le 100) \vee$$
$$(\exists i,b,t,a,s1,s2{:}\, t \ne i \wedge s1 = s2 \wedge ti + a = 100 \wedge 0 \le a \le i - 1 \wedge 1 \le i \le 100)$$

≡

$$(\exists i,b,t,a{:}\, t = i \wedge t^2 + a = 100 \wedge 0 \le a \le t - 1 \wedge 1 \le t \le 100) \vee$$
$$(\exists i,b,t,a,s1,s2{:}\, t \ne i \wedge s1 = s2 \wedge ti + a = 100 \wedge 0 \le a \le i - 1 \wedge 1 \le i \le 100)$$

Damit hat man 2 lipps, die mit dem Verfahren nicht gelöst werden können. „Von Hand“ kann eine Lösung des zweiten lipp berechnet werden:

$$i = 1 \wedge t = 100 \wedge a = 0 \wedge b(1) = b(100)\,.$$

Ausblick

In dieser Arbeit ging es um die Verifikation sequentieller Programme. Zu diesem Zweck werden VCs und ein automatischer Beweiser zum Beweis der VCs benötigt. Die Generierung exakter VCs für elementare Konstrukte sequentieller Programmiersprachen ist nichts Neues. Für Schleifen im Kontext umgebender Anweisungen und rekursive Prozeduren, für die es nur approximierte VCs gibt, wurden leistungsfähigere VCs vorgestellt. Der Schwerpunkt der Arbeit lag auf der Verifikation von Schleifen, die nur durch ihre Spezifikation und ihren Rumpf gegeben sind. Für For-Schleifen wurde ein Verfahren vorgestellt, das in gewissen Fällen automatisch eine Invariante erzeugt. Für While-Schleifen wurde ein Verfahren vorgestellt, das in gewissen Fällen die schwächste Vorbedingung der While-Schleife berechnen kann. Für Programme, deren Korrektheit nicht bewiesen werden kann, wurden erstmals Falsifikationsbedingungen eingeführt. Damit kann zum einen explizit die Nichtkorrektheit des Programms gezeigt werden und zum anderen ist es möglich, automatisch einen Anfangszustand zu ermitteln, der zu einer verletzten Zusicherung führt. Damit bekommt man viel mehr Information zur Lokalisierung eines Programmfehlers als durch die einfache Aussage „die Korrektheit des Programms kann nicht bewiesen werden“.

Fortschritte in der Verifikation sequentieller Programme werden einerseits durch VCs für weitere Daten- und Anweisungstypen erzielt und andererseits durch Fortschritte in den Verfahren, die die Programmverifikation benutzt. Das wichtigste davon ist automatisches Beweisen. Die in dieser Arbeit vorgestellten Verfahren zur Ermittlung der *wp* für While-Schleifen und zur Ermittlung eines Gegenbeispiels für nicht korrekte Programme benutzen die *E*-Unifikation und Methoden des *integer programming*. Die 3 Verfahren zum Beweis einer VC, zur Ermittlung der *wp* für While-Schleifen und zur Ermittlung eines Gegenbeispiels für nicht korrekte Programme haben eins gemeinsam: sie sind partiell in dem Sinne, daß sie als Eingabe ein Problem bekommen und die Ausgabe aus einer Lösung des Problems besteht oder der Angabe, daß das Problem wegen der Begrenztheit des Verfahrens nicht gelöst werden kann.

Ein großer Flaschenhals des in FPP verwendeten Beweisers *mexana* ist die Erzeugung einer DNF im Rahmen der Generierung der Klauseln. Es hat sich herausgestellt, daß schon bei relativ kleinen VCs sehr viele Klauseln entstehen können, von denen die meisten gleich sind. D.h. es gibt nur relativ wenige verschiedene Klauseln. Nützlich wäre hier eine DNF-Routine, die keine gleichen Disjunkte erzeugt. Das könnte man dadurch erreichen, daß man jedes neu erzeugte Disjunkt daraufhin untersucht, ob es schon vorhanden ist. Doch diese Vorgehensweise bringt nicht viel, da immer noch alle Disjunkte erzeugt werden müssen. Man braucht eine DNF-Routine, die von vornherein nur verschiedene Disjunkte erzeugt.

Jeder Datentyp besteht aus der Menge der Werte und den darauf definierten Operationen. Um VCs mit Variablen eines Datentyps beweisen zu können, müssen die für diesen Datentyp gültigen Axiome benutzt werden. Der hier verwendete Beweiser benutzt Axiome für ganze Zahlen und Wahrheitswerte, so daß die Datentypen `integer` und `boolean` behandelt werden können. Um andere Datentypen behandeln zu können, müssen zunächst Axiome für diese Datentypen vorhanden sein. Das gilt z.B. für Fließkommazahlen, Aufzählungstypen, Strings und Verweistypen. Axiome für Aufzählungstypen und Strings sind relativ einfach aufzustellen, da sie nur wenige Operationen haben. Für Fließkommazahlen gibt einige Ansätze zur Axiomatisierung in [IEEE85] und [EP97]. Ob Zeigertypen überhaupt vollständig axiomatisierbar sind, ist nicht klar. Die wenigen Arbeiten auf dem Gebiet der Verifikation mit Pointern sind nicht sehr vielversprechend. Möglicherweise sind Programme mit beliebiger Verzeigerung gar nicht verifizierbar, da die Verifikation eine statische Programmanalyse ist und der Zustand der Verzeigerung etwas sehr dynamisches ist, und zwar viel dynamischer als z.B. der Wert einer Variablen. Die Verifizierbarkeit von Programmen mit beliebiger Verzeigerung ist somit eine

offene Frage, die entweder durch ein Verfahren positiv bzw. durch einen Beweis, daß sie nicht durchführbar ist, negativ beantwortet wird.
Neben einer Axiomatisierung von Datentypen ist es auch erforderlich, daß der verwendete Beweiser in der Lage ist, aus diesen Axiomen ableitbare Sätze zu beweisen. Denn es genügt nicht, dem Beweiser einfach nur Axiome zu geben. Darüber hinaus enthalten die VCs, die aus Programmen mit verschiedenen Datentypen generiert werden, Ausdrücke mit Variablen dieser Datentypen, d.h. gemischte Ausdrücke. Zudem gibt es Funktionen, deren Argumente und Rückgabewert verschiedene Typen haben können.
Neben weiteren Datentypen sollten auch weitere Anweisungstypen, wie z.B. das exit-Statement, und Merkmale moderner Programmiersprachen behandelt werden. Dazu gehören Generizität, Ausnahmebehandlung und Objektorientierung.
Als Fazit kann man sagen, daß es noch ein langer Weg bis zur Verifikation eines realen, vollständigen Programmes ist, verbunden mit der Hoffnung, daß diese Arbeit einen kleinen Schritt auf diesem Weg geleistet hat.

Literatur

[AA78] Alagic, S.: Arbib, M. A.:
The Design of Well-Structured and Correct Programms,
Springer-Verlag New York 1978

[B89] Bijlsma, A.: *Calculating with Pointers*,
Science of Computer Programming 12 (1989) 191-205

[B90] Baber, R. L.: *Fehlerfreie Programmierung für den Software-Zauberlehrling*,
Oldenbourg Verlag München 1990

[B93] Bijlsma, A.: *Calculating with Procedure Calls*,
Information Processing Letters 46 (1993) 211-217

[B95] Best, E.: *Semantik*, Vieweg Verlag Braunschweig 1995

[Ba89] Backhouse, R. C.: *Programmkonstruktion und Verifikation*,
Carl Hanser Verlag München 1989

[BCW90] Bell, C. T.; Cleary, J. G.; Witten, I. H.: *Text Compression*,
Prentice Hall New Jersey 1990

[BMW86] Bijlsma, A.; Matthews, P. A.; Wiltink, J. G.:
Equivalence of the Gries and Martin Proof Rules for Procedure Calls,,
Acta Informatica 23, (1986) 357-360

[BMW89] Bijlsma, A.; Matthews, P. A.; Wiltink, J. G.:
A Sharp Proof Rule for Procedures in wp Semantics,
Acta Informatica 26, (1989) 409-419

[BS92] Baader, F.; Schulz, K. U.:
Unification in the Union of Disjoint