Kapitel 8
Zustandsorientiertes Programmieren

Wir haben in Band I zahlreiche Verfahren kennengelernt, um Algorithmen und Systeme zu beschreiben: Markov-Algorithmen und Semi-Thue-Systeme, endliche Automaten, Petrinetze, Ausdrücke der relationalen Algebra, Termersetzungssysteme, Formeln der Aussagen- und Prädikatenlogik, Schaltfunktionen und Schaltwerke, Entscheidungsdiagramme und Entscheidungstabellen, sowie Programme im λ-Kalkül und in funktionalen Programmiersprachen.

Einige Verfahren beschreiben Beziehungen zwischen verschiedenen Größen; ihre Mächtigkeit reicht, wie bei den Formeln der Prädikatenlogik, weit über das algorithmisch Berechenbare hinaus. Einige überprüfen, ob eine Menge von Größen eine bestimmte Beziehung erfüllt. Andere spezifizieren Abbildungen $f: A \nrightarrow B$ oder deren schrittweise Berechnung.

Bei schrittweiser Berechnung heißen die Größen zusammen mit der Angabe des vorangehenden Schritts der **Zustand** der Berechnung. Die Markierung eines Petrinetzes oder die Angabe, wie weit ein endlicher Automat die Eingabe gelesen, und welchen Automatenzustand er erreicht hat, sind Beispiele dafür. Bei funktionalen Programmen haben wir den Zustandsbegriff in Abb. 5.4 in Abschnitt 5.5.2 gesehen: Die von einem Schritt zum nächsten durchgereichten internen Werte ergeben jeweils den Zustand der Berechnung.

Ein Flipflop bleibt am gleichen Ort und ist damit identifizierbar, unabhängig von seinem Zustand `O` oder `L`. Auch ein Petrinetz ändert seine Struktur nicht, wenn sich seine Markierung ändert. Hingegen sind zwei Flipflops, die unterschiedliche Zustände einnehmen können, voneinander unterscheidbar. Eine Größe, die ihre Identität behält, aber ihren Wert ändern kann, heißt **Zustandsvariable**. Zustandsvariable zusammen mit ihrem Wert ergeben den Zustandsbegriff, wie wir ihn in Abschnitt1.1.1 einführten. Man benutzt häufig die (ungenaue) Sprechweise „der Zustand x“ statt „der Wert der Zustandsvariablen x“.

Zustandsvariable sind allgegenwärtig: Ein Exemplar des vorliegenden Buches ist eine Zustandsvariable, die Zustände wie *neu*, *zerlesen*, *fleckig*, usw. annehmen kann. Betrachten wir nur die Zustandswerte wie im funktionalen Programmieren, ohne den Begriff der Variablen mit gleichbleibender Identität, so können wir den Zustandsübergang *neu* $\rightarrow$ *zerlesen* nicht vom Ersetzen

eines neuen Buchexemplars durch ein anderes, zerlesenes Exemplar unterscheiden. Daß die Zustandsvariable ihre Identität nicht ändert, ist also eine wesentliche, nicht zu vernachlässigende Eigenschaft des Zustandsbegriffs.

Ein Zustand ist gewöhnlich strukturiert und besteht aus Teilzuständen: Der Zustand eines Buches setzt sich aus den Zuständen des Einbands und der Seiten zusammen. Die Teilzustandsvariablen und ihre Werte sind oft selbständige Einheiten, die in einem gegebenen Zusammenhang nur ausschnittweise interessant sind. Für die Beurteilung, ob ein Buch Flecken hat, ist der gedruckte Inhalt der Seiten unerheblich, obwohl er zum Zustand der Seiten, und in einem anderen Zusammenhang auch zum Zustand des Buches, gehört. Wenn wir einen Zustand in dieser Weise als strukturiert ansehen, so nennen wir ihn ein **Objekt**, das aus **Teilobjekten** besteht. Der Begriff *Objekt* ist also rekursiv.

Der Zustand ändert sich durch die Ausführung von Anweisungen. Sprachen, in denen solche Anweisungen formuliert werden, heißen **imperative Programmiersprachen**. Daher spricht man statt von zustandsorientiertem Programmieren gewöhnlich von **imperativem Programmieren**. Die meisten heutigen objekt orientierten Sprachen, darunter auch die in diesem Buch benutzte Sprache SATHER, sind zugleich imperative Sprachen.

Wir benutzen zunächst nur die imperativen Eigenschaften. Auf objektorientierte Eigenschaften gehen wir ab Abschnitt 9.3 ein.

Wir müssen zwangsläufig die Syntax und Semantik einer bestimmten Programmiersprache benutzen, um Programme zu notieren. Der Leser unterscheide die grundlegenden Elemente und Begriffe von ihrer speziellen Ausprägung in der Sprache SATHER, deren Eigenschaften wir in Anhang C zusammengestellt haben. Jede solche Sprache ist ein Artefakt, ein künstlich geschaffenes System, das einen Kompromiß zwischen Ausdrucksmächtigkeit für verschiedene Programmiermethoden und -stile, leichter Erlernbarkeit, einfacher Implementierung und theoretischer Sauberkeit der Begriffsbildung darstellt. Die syntaktischen und semantischen Eigenschaften einer Programmiersprache können die Anwendung bestimmter Programmiermethoden und Konstruktionsprinzipien erleichtern oder erschweren. Die Prinzipien selbst sind jedoch unabhängig von der speziellen Formulierung und bilden den eigentlichen Inhalt dieses und des nachfolgenden Kapitels. Um Gemeinsamkeiten herauszuarbeiten, verweisen wir an zahlreichen Stellen auch auf andere Sprachen.

8.1 Grundbegriffe

Wir stellen die Grundbegriffe des zustandsorientierten Programmierens, nämlich (zustandsorientierte) Variable, bedingte Anweisungen, Schleifen und Prozeduren vor und konzentrieren uns dabei auf den Programmablauf.

8.1.1 Variable und Konstante

> *Angaben können sehr verschiedenartig sein, z. B. Zahlen, Aussagen, Namen, Adressen, Signale, Dienstgrade, Ordinaten, usw. Jede Angabe hat einen Inhalt. Der Inhalt ist das, was mit der betreffenden Angabe gesagt werden soll. Man kann die Angabe in einen* konstanten *und einen* variablen Teil *zerlegen. Den Unterschied macht man sich am besten an dem Beispiel eines Formulars oder Fragebogens klar. Ein solcher Fragebogen enthält vorgedruckte Teile mit Leerstellen, welche individuell auszufüllen sind. Z. B.: „Name . . . " „Geburtsdatum . . . " „verheiratet . . . " usw.*
>
> *Das Vorgedruckte entspricht dem konstanten Teil der Angabe und die Leerstellen dem variablen Teil. Solange die Leerstellen nicht ausgefüllt sind, hat man es mit unbestimmten Größen oder allgemein mit „Variablen" zu tun.* Konrad Zuse, 1944[1]

In logischen und funktionalen Sprachen ebenso wie in der Algebra und Logik sind Variable unbestimmte Werte, die durch eine Substitution einen bestimmten und dann unveränderlichen Wert erhalten. Im zustandsorientierten Programmieren ist eine **Variable** ein Tripel (Referenz, Behälter, Wert). Ein **Objekt** im Sinne des vorigen Abschnitts ist eine strukturierte Variable, deren Wert ein Tupel ist. Die **Referenz** ist ein unveränderliches, eindeutiges Kennzeichen der Variablen; wir nennen sie auch einen **Verweis** auf die Variable oder die **Identität der Variablen**. Der **Behälter** ist die eigentliche Variable, sein Inhalt ist ihr **Wert**. Eine Variable heißt eine **Konstante**, wenn ihr Wert unveränderlich ist. Wir sprechen im folgenden von einer **Größe**, wenn wir zwischen Variablen und Konstanten nicht unterscheiden wollen.

Auf Variable kann man lesend oder schreibend, auf Konstante nur lesend zugreifen. Dazu benötigt man eine **Zugriffsfunktion**, deren Aufruf die Referenz berechnet und dann entweder den Wert liefert oder diesen, bei Variablen, durch einen neuen Wert ersetzt. Die Zugriffsfunktion muß man explizit im Programm als Text wiedergeben können. Diese textuelle Repräsentation einer Zugriffsfunktion nennen wir einen **Namen** (der Variablen oder Konstanten).

Im Rechner entspricht dem Behälter ein Speicherplatz; dessen Adresse repräsentiert oft die Referenz. Adressen sind Werte, die man kopieren kann; man kann also mehrere Referenzen auf die gleiche Variable in verschiedenen Verweisvariablen speichern. Variable und Objekte können während des Ablaufs eines Programms ihren Ort ändern oder sogar in mehreren identischen

1. Konrad Zuse, 1910 – 1995, konstruierte 1941 den ersten programmgesteuerten Digitalrechner Z3 (Nachbau im Deutschen Museum) mit über 2000 Relais. Das Zitat stammt aus der Vorrede des 1944/45 entworfenen Plankalküls, (Zuse, 1972), vgl. auch (Bauer und Wössner, 1972). Der Plankalkül war die erste höhere Programmiersprache und enthielt bereits Zuweisungen, bedingte Anweisungen und Schleifen, aber keine Sprünge. Er wurde allerdings niemals praktisch benutzt und erst 1972 veröffentlicht.

Kopien vorhanden sein, von denen wahlweise eine benutzt wird. Eine Referenz ist also eine begriffliche Abstraktion, die über den Begriff *Adresse* hinausgeht.

Namen für Variable sind im einfachsten Fall **Bezeichner** wie i, x1, bezeichner, auch_dies_ist_ein_Bezeichner. Solche Bezeichner können vom Programmierer frei gewählt werden.

Namen für Konstante sind Zahlen wie 1, 23758, 0.1, 34.687, 1E1, 3.5E+6, 3.5e-6; oder die booleschen Werte true und false; oder Zeichen wie 'c', '#', '␣'; oder Texte wie "dies ist ein Text" oder der leere Text "". Sie repräsentieren Konstante mit dem durch ihren Namen gegebenen Wert. Solche expliziten Konstante heißen **Literale**.

Die grundlegende Operation auf Variablen und die zentrale **Anweisung** zur Änderung des Zustands einer Berechnung ist die **Zuweisung**[2]

Variablenname := *Ausdruck*

Eine Zuweisung berechnet den Wert w des Ausdrucks auf der rechten Seite und ersetzt den Wert der Variablen auf der linken Seite durch w. Nach einer Zuweisung $v := a$ gilt eine Aussage $Q = Q[v]$ über die Variable v, wenn $Q[a/v]$ vor Ausführung der Zuweisung galt: Es gilt $Q{:}\,x > y$, wenn vor der Zuweisung $y := 7$ die Beziehung $Q[7/y]{:}\,x > 7$ richtig war. $Q[a/v]$ bezeichnet dabei wie gewohnt den Ausdruck, den man aus Q erhält, indem man überall den Wert von a anstelle von v einsetzt.

Eine Variable im zustandsorientierten Programmieren verhält sich also wie eine Größe v in einer funktionalen Sprache: Auch dort gilt eine Aussage $Q[v]$, wenn es einen Ausdruck a gibt, so daß $v = a$ eine Definition im Programm und $Q[a/v]$ gültig ist.

Allerdings ordnet im funktionalen Programmieren die Definition $v = a$ der Größe v den Wert a ein für allemal zu. Diese Situation treffen wir auch im zustandsorientierten Programmieren an. v heißt dann eine **benannte Konstante** oder eine Variable mit **Einmalzuweisung**[3]. Sie wird durch eine Konstantenvereinbarung

```
constant pi : FLT := 3.1415926; -- Gleitpunktkonstante
```

eingeführt.

Im allgemeinen Fall kann es zu einer Variablen v im zustandsorientierten Programmieren mehrere Zuweisungen geben, die v zu unterschiedlichen Zeiten verschiedene Werte zuordnen. Abhängig vom jeweiligen Wert von v kann daher ein Ausdruck wie $v + 1$ verschiedene Ergebnisse liefern. Die Zuweisung

```
v := v + 1
```  (8.1)

bedeutet

neuer Wert von v := (alter Wert von v) + 1

2. In manchen Sprachen, z. B. in FORTRAN, heißt eine Zuweisung eine Definition.
3. engl. *single assignment property*.

Hat v anfangs den Wert 0, so führt die wiederholte Ausführung von (8.1) zu $v = 1$, $v = 2$, ... Die wiederholte Ausführung derselben Zuweisung $v := a$ ist also sinnvoll: Der Variablen v werden die unterschiedlichen Werte des Ausdrucks a zugewiesen.

Neben dem eigentlichen Wert w einer Variablen v, also z. B. einer Zahl oder einem Text, kann in bestimmten Fällen auch die Referenz der Variablen v als Wert betrachtet werden: Ein Vergleich $u = v$ könnte alternativ die Frage „besitzen u und v augenblicklich den *gleichen* Wert?“ oder die Frage „sind u und v Zugriffsfunktionen für *dieselbe* Variable?“ beantworten. Natürlich muß man diese beiden Fälle unterscheiden, z. B. indem man den Vergleichsoperator unterschiedlich notiert, oder indem man explizit angibt, wann u bzw. v den Wert der Variablen und wann sie die Referenz darstellen. Hierfür hat sich bis heute kein einheitlicher Lösungsvorschlag durchsetzen können. Wir stellen die Erörterung der Verwendung von Referenzen als Werte einstweilen zurück.

In der Programmiersprache C sind die Referenz und der Wert als *left hand value* und *right hand value* einer Variablen bekannt. Die Begriffe gehen auf die Sprache CPL von C. Strachey[4] zurück, aus der auch viele andere Eigenschaften von C stammen. Mit *Linkswerten*, also mit Referenzen, kann man in C ähnlich wie mit ganzen Zahlen rechnen. Dies hat sich als eine der mächtigsten, zugleich aber auch als die gefährlichste Eigenschaft von C erwiesen.

8.1.2 Vereinbarungen, Programme

Aus den in Abschnitt 5.4 genannten Gründen unterteilen funktionale Sprachen die Menge der möglichen Werte und ordnen sie einzelnen abstrakten Datentypen zu. In Abschnitt 5.4.1 sahen wir, daß man den Typ eines Bezeichners oder, allgemeiner, eines Ausdrucks, durch Typinferenz bestimmen kann. Eine Vereinbarung des Typs eines Bezeichners dient lediglich der Kontrolle; funktionale Sprachen sind stark typgebunden.

Beim zustandsorientierten Programmieren scheitert Typinferenz in vielen Fällen, da während einer Übersetzung nicht alle Aufrufe einer zu übersetzenden Funktion und daher auch nicht alle Zuweisungen an eine Variable bekannt sein müssen.

Die meisten imperativen Sprachen verlangen daher, daß sämtliche Bezeichner für Größen durch eine **Vereinbarung** *bezeichner* : *Typ* eingeführt werden, also z. B.

```
    v: INT
```

oder

```
    i,j: INT
```

oder

```
    wert: INT := 0
```

4. Christopher Strachey, 1916 – 1975, Professor der Informatik an der Universität Oxford.

Das zweite Beispiel faßt die Vereinbarungen i: INT und j: INT zusammen. Das dritte Beispiel verknüpft die Vereinbarung mit einer initialen Zuweisung des Wertes 0. Man spricht von einer Vereinbarung mit **Vorbesetzung** oder **Initialisierung**. Durch zusätzlichen Gebrauch des Schlüsselworts constant erhält man die Konstantenvereinbarung aus dem vorigen Abschnitt.

In SATHER schreiben wir Typbezeichner wie INT mit Großbuchstaben. Diese Konvention is willkürlich. In C, JAVA und C# schreibt man stattdessen int, char usw. Außerdem notieren diese Sprachen den Typ vor dem Variablenbezeichner, also int i statt i: INT. Konstante können in C++ und C# mit dem Schlüsselwort const eingeführt werden. Konstante in JAVA kennzeichnet das Schlüsselwort final; sie sind jedoch nur an bestimmten Stellen zugelassen.

Der Wert einer Variablen v, deren Vereinbarung keine Vorbesetzung enthält, ist bis zur ersten Zuweisung $v := a$ unbekannt und darf in einem korrekten Programm nicht benutzt werden. Der Programmierer muß diese Bedingung für nicht-vorbesetzte Variable sicherstellen.

Um diese Fehlerquelle zu umgehen, werden in C, C++ und JAVA statisch allozierte Variable ohne explizite Vorbesetzung implizit mit dem Wert 0 oder allgemein mit dem Wert, dessen binäre Codierung nur aus 0 besteht, vorbesetzt. In C# wird mit Ausnahme lokaler Variablen ebenfalls diese Vorbesetzung gewählt (lokale Variablen haben keine Vorbesetzung). Diese Lösung verschiebt aber nur das Problem: der Programmierer muß jetzt sicherstellen, daß 0 ein akzeptabler Anfangswert ist.

Durch **Prozedurvereinbarungen**

```
procedure prozedurname : e_Typ is ... end
procedure prozedurname (parameter_name : p_Typ; ... ) :e_Typ is ... end
```

oder

```
procedure prozedurname is ... end
procedure prozedurname (parameter_name : p_Typ; ... ) is ... end
```

werden Bezeichner zur Benennung von **Prozeduren** eingeführt. Die erste Form vereinbart **Funktionsprozeduren**, die wie Funktionen funktionaler Sprachen ein Ergebnis des **Ergebnistyps** *e_Typ* liefern. Falls Parameter vorhanden sind, bezeichnet *p_Typ* den Typ eines Parameters. Eine Verwendung von prozedurname in einem Ausdruck heißt ein **Funktionsaufruf**. Zur Unterscheidung von Funktionen heißen Prozeduren ohne Ergebnistyp auch **eigentliche** oder **echte Prozeduren**[5].

Vor dem Symbol is steht der **Prozedurkopf**, der die Parameter und Ergebnisse, also die **Signatur** der Prozedur spezifiziert. Nach is folgt der **Prozedurrumpf**, die eigentliche Rechenvorschrift. In SATHER kann man das einleitende Schlüsselwort procedure auch weglassen; davon machen wir im folgenden aus Platzgründen Gebrauch. In Kap. 10 erläutern wir, warum Prozeduren auch **Methoden**[6] heißen.

5. engl. *proper procedure*.

6. Dieser Wortgebrauch hat sich für objektorientierte Programmiersprachen ausgehend von SMALLTALK eingebürgert. Etwa in der Zusammensetzung *Zerlegungsmethode* bedeutet *Methode*

Auf weitere Eigenschaften von Prozeduren gehen wir in Abschnitt 8.1.6.4 ein.

Ein **Programm** in einer imperativen Sprache ist genau wie in funktionalen Sprachen eine Menge von Vereinbarungen von Größen und Methoden. In SATHER schreiben wir im einfachsten Fall

```
class Programmname is
    Vereinbarung_1;
    ...
    Vereinbarung_n
end
```

Während beim funktionalen Programmieren Ausdrücke vorgegeben und mit Hilfe des Programms berechnet werden, wird beim imperativen Programmieren eine der Prozeduren des Programms als **Hauptprogramm** ausgezeichnet: die Programmausführung besteht aus der Ausführung des Hauptprogramms.

In SATHER, C, C++, JAVA und C# hat das Hauptprogramm den Namen main; das Programm besteht aus allen Vereinbarungen, die direkt oder indirekt bei der Ausführung von main benötigt werden. In C ist keine feste syntaktische Struktur für Programme vorgeschrieben. In SATHER, JAVA und C# gehören die Vereinbarungen zu einer oder mehreren Klassen. Das Programm ist eine Menge von Klassen, von denen eine das Hauptprogramm enthält und dadurch ausgezeichnet ist. In PASCAL (und ähnlich in MODULA-2) schreibt man program *Programmname*(...); *Vereinbarungen* begin ... end; begin ... end ist der (unbezeichnete) Rumpf des Hauptprogramms.

8.1.3 Gültigkeitsbereich und Lebensdauer

In Abschnitt 5.2.2.1 hatten wir den **Gültigkeitsbereich**[7] einer Vereinbarung eines Bezeichners *b* als den Teil eines Programmtexts definiert, in dem Anwendungen von *b* die Bedeutung aus der gegebenen Vereinbarung haben. Diese Definition übernehmen wir auch für imperative Sprachen. Solange man Vereinbarungen und ihre Anwendung gleichzeitig ersetzt und keine Namenskonflikte auftreten, sind Bezeichner beliebig austauschbar. Die Wahl der Bezeichner ist ausschließlich eine Frage der Verständlichkeit des Programmtexts für den menschlichen Leser.

Gültigkeitsbereiche in funktionalen Sprachen können ineinander geschachtelt sein. Lokale Vereinbarungen können globale Vereinbarungen des gleichen Bezeichners verdecken. Die Regeln hierfür entsprechen den zulässigen Substitutionen aus Abschnitt 5.1.4 und den Bindungsregeln der Prädikatenlogik in Abschnitt 4.2.1.

Dieselben Regeln gelten auch in imperativen Sprachen: In SATHER sind **Blöcke**, d. h. Anweisungsfolgen mit (möglicherweise) vorangestellten Vereinbarungen, Methoden und Klassen ineinander geschachtelte Gültigkeitsbereiche;

natürlich etwas anderes. In C, C++, JAVA und C# heißen eigentliche und Funktionsprozeduren einheitlich Funktionen; eine eigentliche Prozedur ist eine Funktion, deren Ergebnis den leeren Typ void hat, von dem es keine Werte gibt.

7. engl. *scope*.

es gilt jeweils die Vereinbarung aus dem kleinsten Gültigkeitsbereich, der die gegebene Anwendung eines Bezeichners umfaßt. Dabei vereinbart eine Methode nach außen sichtbar den Methodennamen mit seiner Signatur als Typ. Ihr Rumpf ist ein Block, zu dem auch etwaige Parametervereinbarungen gehören.[8]

Beispiel 8.1: Wir nehmen an, daß in dem Programmschema

```
class Programmname is
  a: INT;
  procedure p is a: INT; ··· a ··· end;
  procedure q(a: INT) is ··· a ··· end;
  procedure r is ··· a ··· end;
  procedure s(a: INT) is a: INT; ··· a ··· end; -- unzulässig
end;
```

nur die angeschriebenen Vereinbarungen vorkommen. Dann bezieht sich der Bezeichner *a* im Rumpf von *p* auf die lokal vereinbarte Variable, im Rumpf von *q* auf den Parameter und nur im Rumpf von *r* auf die global vereinbarte Variable. Die zweite Vereinbarung von *a* im der Prozedur *s* ist unzulässig, da die Parameterspezifikation als Vereinbarung im Prozedurrumpf gilt. Zwei Vereinbarungen des gleichen Bezeichners in einem Block sind aber nicht erlaubt.

In C gelten gleichartige Regeln, wenn man den Begriff *Klasse* durch *getrennt übersetzbare Programmeinheit* ersetzt. In Java und C# darf in geschachtelten Blöcken der gleiche Bezeichner nicht nochmals vereinbart werden.

An der Prozedur *r* sehen wir, daß im Gegensatz zum funktionalen Programmieren mehrfaches Aufrufen einer parameterlosen Prozedur unterschiedliche Ergebnisse zeitigen kann: Die Änderung des Werts der globalen Größe *a* während oder zwischen Aufrufen von *r* kann das Ergebnis beeinflussen; die Wirkung eines Prozeduraufrufs ist abhängig vom Zustand zum Zeitpunkt des Aufrufs. ♦

Zusätzlich kann es Bezeichner geben, die in gewissen syntaktischen Strukturen automatisch mit einer bestimmten Bedeutung erklärt sind. So wird in Sather der Bezeichner res[9] automatisch in jeder Funktionsprozedur als eine Variable mit dem Ergebnistyp der Prozedur vereinbart. Ihr Wert ist am Ende der Ausführung der Prozedur der Funktionswert.

In funktionalen Sprachen kümmern wir uns nicht weiter um die Werte von Größen, deren Gültigkeitsbereich wir verlassen. Soweit sie Speicher belegen, z. B. umfangreiche Listen, ist es Sache der Implementierung der Sprache, den Speicher zu bereinigen und Platz für andere Werte zu schaffen. Dieser Vorgehensweise liegt der (mathematische) Gedanke zugrunde, daß Werte „von Ewigkeit zu Ewigkeit“

8. Die Erfindung des Blockschachtelungsprinzips wird Klaus Samelson, 1918-1980, Professor der Informatik in München, zugeschrieben.

9. für *Resultat*.

existieren, aber nur während der Zeit, in der ihnen ein Bezeichner zugeordnet ist, in unser Gesichtsfeld treten.

Im zustandsorientierten Programmieren ist die Ausführung einer Vereinbarung einer Größe zugleich eine Zustandsänderung: der Zustand wird um eine Zustandsvariable, nämlich die vereinbarte Größe, erweitert. Sobald diese Zustandsvariable nicht mehr zugreifbar ist, endigt ihre Lebensdauer; sie wird aus dem Gesamtzustand gestrichen. Die **Lebensdauer**[10] einer vereinbarten Größe, oder kurz, die Lebensdauer einer Vereinbarung, ist also derjenige Ausschnitt eines Programm*ablaufs* in dem die vereinbarte Größe existiert. Ein Gültigkeitsbereich ist ein Teil eines (Programm-)Textes; die Lebensdauer bezieht sich hingegen auf dessen Ausführung. Man vergleiche dazu Abschnitt 1.3; die hier als Größen auftretenden Elemente hatten wir dort als zum System gehörige Gegenstände oder Bausteine bezeichnet.

8.1.4 Typen und Operationen

Imperative Sprachen stellen zur Verarbeitung von Zahlwerten, Zeichen usw. die einfachen Datentypen in Tab. 8.1 mit unterschiedlichen Bezeichnungen zur Verfügung; wir hatten sie bereits bei funktionalen Sprachen kennengelernt. Mit

Tabelle 8.1: Einfache Datentypen

Typ	Grundoperationen (Auswahl)	Beschreibung
BOOL	and, or, not	BOOL $\triangleq$ $\mathbb{B}$
INT	+, −, ∗, div, mod	Ganzzahlen beschränkter Größe
FLT, FLTD	+, −, ∗, /, ^	Gleitpunktzahlen beschränkter Genauigkeit
CHAR	pred,succ	Einzelzeichen (ASCII, ISO 8859-1, UNICODE, . . .)
STR	*nicht allgemein festgelegt*	Texte (Folge v. Einzelzeichen)

allen diesen Typen kann man Variablen- und Konstantenvereinbarungen bilden. Wenn wir Ausdrücke F unter Verwendung von Operationen aus der Tabelle oder anderer Operationen bilden, bezeichnen wir das Ergebnis der Berechnung des Ausdrucks in einem Zustand z mit $z : F$. Das Ergebnis existiert wie bei Ausdrücken in funktionalen Sprachen nur, wenn

- das Ergebnis nach Regeln des abstrakten Datentyps, zu denen die Operanden und Operationen gehören, existiert; Division durch 0 ist also nicht erlaubt;

10. engl. *extent* oder *life-time*.

- das Ergebnis arithmetischer Ausdrücke im Rahmen der beschränkten Genauigkeit der Rechnerarithmetik berechenbar ist;
- alle Operanden im Zustand z einen Wert besitzen (alle Variablen vorbesetzt) und der Gleitpunktwert NaN ("Not a Number", vgl. Bd. I, Anhang B.2.2) nicht vorkommt.

Bei Erfüllung dieser Bedingungen heißt der Ausdruck F **zulässig**. Wir kennzeichnen ihn mit dem Prädikat zulässig(F).

Unter den zahlreichen hier nicht genannten Operationen finden sich insbesondere die Vergleichsoperationen $=, \neq : \text{Typ} \times \text{Typ} \to \text{BOOL}$ und zumindest für die numerischen Typen INT, FLT und den Typ CHAR die Vergleichsoperationen $\leqslant, <, >, \geqslant : \text{Typ} \times \text{Typ} \to \text{BOOL}$. Falls die Vergleichsoperationen $\leqslant, \ldots$ auch für Texte definiert sind, benutzen sie die lexikographische Ordnung entsprechend der Codierung des Zeichensatzes: "" < "a", "abc" < "abd", "abc" < "abcd".

Die booleschen Operationen and und or unterscheiden sich in vielen Programmiersprachen von den anderen Operationen: Während bei $a \ \tau \ b$ beide Operanden berechnet und dann die Operation τ angewandt wird – bei funktionalen Sprachen bezeichneten wir dies als strikte Berechnung – wird a and b und a or b oft so definiert, daß der zweite Operand nur berechnet wird, wenn der erste das Ergebnis noch nicht festlegt. In funktionaler Schreibweise gilt also

$$\begin{array}{lcl} \text{Ergebnis}(a \text{ and } b) & = & \text{if } a \text{ then } b \text{ else false} \\ \text{Ergebnis}(a \text{ or } b) & = & \text{if } a \text{ then true else } b \end{array} \tag{8.2}$$

Diese faule Berechnung heißt **sequentielle** oder **Kurzauswertung** (eines booleschen Ausdrucks)[11]. Sie ist notwendig für Ausdrücke wie

$$x \neq 0 \text{ and } (1 - x)/x > 1, \tag{8.3}$$

in denen die Erfüllung der ersten Teilbedingung Voraussetzung für die Berechnung der zweiten ist.

Die Operationen div und mod liefern den Quotienten und den Rest. Für $a, b \neq 0$ gilt

$$a = (a \text{ div } b) * b + a \text{ mod } b, \quad 0 \leqslant |a \text{ mod } b| < |a|. \tag{8.4}$$

Technisch liefert die Divisionsoperation den Rest meist mit dem Vorzeichen des Zählers. Für $a \text{ mod } b \neq 0$ gilt daher $\text{sign}(a) = \text{sign}(a \text{ mod } b)$ mit der Vorzeichenfunktion sign(x). Wie bereits in Abschnitt 5.3.2 bemerkt, ist es zweckmäßig, die Restoperation nur mit positiven Operanden zu benutzen.

Gleitpunktzahlen gibt es in verschiedenen Ausprägungen, die sich durch die Länge der Mantisse und des Exponenten unterscheiden: In SATHER bezeichnet FLT den Typ von Gleitpunktzahlen einfacher und FLTD den Typ von Gleitpunktzahlen doppelter Länge entsprechend dem IEEE-Standard, vgl. Anhang B.2; insbesondere gibt es für diese Typen undefinierte Werte NaN und den Wert Inf, d. h. ∞.

11. engl. *short-circuit evaluation*.

Viele Sprachen verlangen, daß in einer Zuweisung $a := b$ oder einem Ausdruck $a + b$ alle Größen einen einheitlichen (numerischen) Typ haben. SATHER gehört zu den Sprachen, in denen es eine automatische **Typanpassung**[12]

$$\mathsf{INT} \to \mathsf{FLT} \to \mathsf{FLTD} \tag{8.5}$$

gibt: Wenn ein Wert b des Typs T einer Variable a eines in dieser Reihe nachfolgenden Typs T' zugewiesen oder mit deren Wert verknüpft wird, wird der Wert w automatisch in einen Wert b' des Typs T' umgewandelt. Die umgekehrte Typanpassung FLT bzw. FLTD $\to$ INT wird heute grundsätzlich explizit geschrieben, um die Art der Rundung zu kennzeichnen. In den meisten imperativen Sprachen gibt es dafür einstellige Funktionen wie truncate(x) oder round(x). In SATHER schreiben wir solche Funktionen nachgestellt: x.int liefert die nächste ganze Zahl, die dem Betrag nach nicht größer ist als x. x.round ergibt die nächstliegende ganze Zahl.[13] Die Signaturen der beiden Funktionen sind

$$\text{int:} \quad \mathsf{FLT} \to \mathsf{INT}, \tag{8.6}$$

$$\text{round:} \quad \mathsf{FLT} \to \mathsf{INT}. \tag{8.7}$$

Sie sind auch für den Typ FLTD definiert. Es gilt

$$0 \leqslant |x.\mathsf{int}| \leqslant |x| < |x.\mathsf{int}| + 1 \text{ und } \operatorname{sign}(x.\mathsf{int}) = \operatorname{sign}(x), \text{ falls } x.\mathsf{int} \neq 0 \tag{8.8}$$

sowie

$$x.\mathsf{round} = (x + 0.5\mathit{sign}(x)).\mathsf{int}. \tag{8.9}$$

Aufgabe 8.1: Definieren Sie mit Hilfe von int die (in SATHER vorhandenen) Operationen x.ceiling $= \lceil x \rceil$ und x.floor $= \lfloor x \rfloor$ mit ganzzahligem Ergebnis und $\lceil x \rceil - 1 < x \leqslant \lceil x \rceil$ bzw. $\lfloor x \rfloor \leqslant x < \lfloor x \rfloor + 1$.

Eine Vereinbarung c: CHAR führt eine Variable für Einzelzeichen ein. Die einstelligen Operationen pred und succ, in SATHER nachgestellt geschrieben, c.pred bzw. c.succ, liefern das vorangehende bzw. nachfolgende Zeichen in der Reihenfolge der Codierung. Daneben gibt es noch weitere Operationen, die zwischen Typen konvertieren:

$$\begin{aligned} \text{int:} &\quad \mathsf{CHAR} \to \mathsf{INT} \\ \text{char:} &\quad \mathsf{INT} \to \mathsf{CHAR} \\ \text{str:} &\quad \mathsf{CHAR} \to \mathsf{STR}. \end{aligned}$$

Natürlich kann man nur Zahlen i zwischen 0 und der maximalen Codierung, also 255 oder bei Verwendung von 16 bit UNICODE entsprechend 65535 als Zeichen interpretieren.

12. engl. *coercion*.

13. Man beachte, daß beim Übergang INT $\to$ FLT Rundungsfehler auftreten können, wenn wir 32 Bit für ganze Zahlen und einfach genaue Gleitpunktzahlen in der Codierung von Bd. I, Abb. B.5 zugrundelegen. Umgekehrt können x.round und x.int Überlauf hervorrufen.

Der Typ STR kennzeichnet Texte, also Folgen von Zeichen. Eine Textkonstante, vereinbart durch

constant titel: STR := "Vorlesungen über Informatik"

besteht aus einer festen Anzahl von Zeichen, hier 27; die Anzahl wird der Vorbesetzung entnommen. Eine Textvariable *tv*: STR[*n*] hat eine feste Obergrenze *n* und kann Texte der Länge *l*, $0 \leqslant l < n$ aufnehmen. In SATHER liefert nach der Vereinbarung der Ausdruck *tv*.asize den Wert *n*. Die aktuelle Länge *l* ergibt sich zu *tv*.length.

Eine Textvariable hat als Wert also ein Tupel (Obergrenze, Länge, Text). In C und ebenso in SATHER und anderen Sprachen wird auf die Längenangabe verzichtet. Sie wird dem Text entnommen: Das erste Auftreten des Zeichens NUL = 0.char im Text beendet den Text und bestimmt seine Länge (über die weiteren Zeichen wird keine Aussage gemacht!). NUL ist also kein zulässiges Zeichen in einem Text, sondern dient als **Anschlag**[14], der den Text abschließt.

STR[n] ist in Wahrheit eine Kurzschreibweise für den Spezialfall ARR[*n*](CHAR) einer **dynamischen Reihung**, vgl. Abschnitt 6.2.2. Reihungen können wir für Elemente beliebigen Typs *T* (nicht nur CHAR) definieren. Eine Größe *a*: ARR[*n*](*T*) ist eine geordnete Menge von Variablen mit gemeinsamem Namen *a*. Die Einzelwerte werden durch Indizierung ausgewählt. In mathematischer Notation ist $a = \{a_0, a_1, \ldots, a_{n-1}\}$. In Programmiersprachen schreiben wir $\{a[0], a[1], \ldots, a[n-1]\}$. Die ganzen Zahlen i, $0 \leqslant i < n$ bilden die **Indexmenge** $\mathcal{I}(a)$ der Reihung *a*. Die Obergrenze *n* bezeichnen wir wie bei Texten mit *a*.asize. Die Vereinbarung *a*: ARR[3](FLT){0.5, – 1.0, 1.5} definiert die Reihung *a* mit den drei Variablen $a[0] = 0.5$, $a[1] = -1.0$, $a[2] = 1.5$ und $a.\text{asize} = 3$. Eine Zuweisung $a[1] := 2.1$ ändert die einzelne Variable $a[1]$ und damit natürlich zugleich den Wert von ganz *a*. Es gilt danach $a = \{0.5, 2.1, 1.5\}$.

Damit eine Zuweisung $a[i] := 2.1$ oder die Verwendung von $a[i]$ als Operand möglich ist, muß $i \in \mathcal{I}$, also $0 \leqslant i < a.\text{asize}$, gelten. Diese Bedingung fügen wir bei Formeln *F* dem Prädikat zulässig(*F*) konjunktiv hinzu.

Auch einer Textvariablen, vereinbart mit *tv*: STR[3], können wir den Text "abc" in der Schreibweise *tv* := {'a', 'b', 'c'} zuweisen. Ferner ersetzt *tv*[1] := 'd' den Text durch "adc". Auch Texte können indiziert werden.

In Sprachen wie PASCAL vereinbart *a*: **array**[2 . . 5] **of** real eine Reihung mit der Indexmenge $\mathcal{I}(a) = \{2, 3, 4, 5\}$; man kann nicht nur die Obergrenze, sondern auch die Untergrenze der Indexmenge frei wählen. In der Mathematik ist die Untergrenze 1 üblich; diese Konvention wurde in FORTRAN übernommen. In SATHER ist ebenso wie in C, C++, JAVA und C# die Untergrenze stets 0. Diese Konvention hat technische Gründe, vgl. Kap.11, ist aber auch für viele praktische Anwendungen sehr bequem.

Eine Abbildung $f: X \nrightarrow Y$ zwischen endlichen Mengen können wir als eine Menge von Paaren $i \mapsto f(i)$ ansehen. Nehmen wir die Indexmenge $\mathcal{I}$ einer

14. engl. *sentinel*.

Reihung a als Definitionsbereich X und die Werte $a[i]$, $i \in X$, als Funktionswerte $f(i)$, so erweist sich die Reihung a mit Elementen vom Typ T als eine Abbildung $a\colon \mathcal{I} \to T$. Eine Zuweisung $a[i] := t$ verändert diese Abbildung an der Stelle i. Für die neue Abbildung a' gilt

$$a'[j] = \begin{cases} t, & j = i, \\ a[j], & j \neq i. \end{cases} \tag{8.10}$$

Wir bezeichnen diese neue Abbildung kurz mit $a' = a : [t/i]$. Die Ähnlichkeit mit der Notation für Substitutionen ist beabsichtigt: $a : [t/i]$ ist die Abbildung, die aus der Menge der Substitutionen $[a[j]/j]$ für $j \neq i$ und der Substitution $[t/i]$ besteht.

Daß eine Reihung eine Funktion ist, ist eine nützliche Abstraktion, auch wenn sie nicht der Realisierung entspricht. Insbesondere können wir bei Bedarf Reihungen auch als Größen eines einfachen Typs, nämlich eines Funktionstyps, beschreiben, obwohl wir Reihungstypen gewöhnlich zu den zusammengesetzten Typen rechnen.

Für Reihungen ist neben der Indizierung und der Gesamtzuweisung sehr häufig die Bildung eines **Abschnitts** $a[i : j]$ wichtig. $a[i : j]$ ist eine neue Reihung mit

$$a[i : j][k] = \begin{cases} a[k], & i \leqslant k \leqslant j, \\ \text{undefiniert}, & \text{sonst}. \end{cases} \tag{8.11}$$

Für $j < i$ bezeichnet $a[i : j]$ die **leere Reihung**, die 0 Elemente enthält. In SATHER schreiben wir $a.\mathsf{subarr}(i, j)$ statt $a[i : j]$; wegen der Festlegung auf die Untergrenze 0 gilt $a.\mathsf{subarr}(i, j)[l] = a[i + l]$ mit $k = i + l$.

In imperativen Sprachen heißen die Tupel (x, y, z) funktionaler Sprachen, bei denen die Werte x, y, z unterschiedlichen Typ haben können, **Verbunde**[15]. In PASCAL wird z. B. durch die Vereinbarung

$\mathsf{type}\ t = \mathsf{record}\ x\colon \mathsf{integer}; y, z\colon \mathsf{real}\ \mathsf{end}$

der Typ eines Verbunds mit drei **Feldern** x, y, z eingeführt. Mit $v\colon t$ kann man anschließend Variable dieses Typs vereinbaren und mit $v.x$, $v.y$, $v.z$ auf die Felder zugreifen. Wir nennen in diesem Zusammenhang v den **Qualifikator** eines **Feldbezeichners** x. $v.x$ heißt ein **qualifizierter Name**.

Ebenso wie Reihungen können wir auch Verbunde als Abbildungen auffassen. Ihr Definitionsbereich ist kein Ausschnitt der ganzen Zahlen, sondern die Menge der Feldbezeichner. Der Wertebereich ist die disjunkte Vereinigung $T_1 \uplus T_2 \uplus \cdots$ der Typen der Felder, vgl. A.2, in unserem PASCAL-Beispiel also $t\colon \{x, y, z\} \to \mathsf{integer} \uplus \mathsf{real} \uplus \mathsf{real}$.

Eine Variable eines Verbundtyps ist eine strukturierte Variable, also ein Objekt im Sinne von Abschnitt 8.1.1. Dies hat *objekt*orientierten Sprachen ihren Namen gegeben. In diesen Sprachen heißen die Felder gewöhnlich **Merkmale**[16];

15. engl. *record*.
16. engl. *feature*.

auch Methoden sind als Merkmale zugelassen. Die zuvor kommentarlos benutzten Notationen wie *a*.asize finden hier ihre Erklärung. Allgemein sind in solchen Sprachen die binären Operationen genauso wie in funktionalen Sprachen durch Curryen definiert. So wird in SATHER $a + b$ als Aufruf *a*.plus(*b*) der für den Typ von *a* definierten einstelligen Funktion plus interpretiert. Im Jargon sagt man „$a + b$ ist syntaktischer Zucker für *a*.plus(*b*)".

8.1.5 Ausdrücke

Wie in funktionalen Sprachen können wir aus einzelnen Operanden (Literale, Bezeichner für Größen, Funktionsaufrufe, qualifizierte Namen, indizierte Namen) und Operatoren Ausdrücke zusammensetzen. Bei Ausdrücken mit mehreren Operatoren kennzeichnen wir Teilausdrücke durch Klammerung. Wie in der Mathematik gibt es Vorrangregeln, um die Anzahl der Klammerpaare zu reduzieren. Die Vorrangregeln für die Operatoren in SATHER zeigt die Tabelle 8.2.

Tabelle 8.2: Vorrangregeln für Operatoren in SATHER

Vorrang	Operation	objektorientierte Funktionsschreibweise	übliche Bedeutung
1	a >> b	*a*.str_in(*b*)	lies b von a
1	a << b	*a*.str_out(*b*)	schreibe b nach a
2	a or b		(faules) $a \vee b$
3	a and b		(faules) $a \wedge b$
4	$a = b$	*a*.is_equal(*b*)	Vergleich $a = b$
4	a /= b	(*a*.is_equal(*b*)).negate	Vergleich $a \neq b$
4	$a < b$	*a*.is_lt(*b*)	Vergleich $a < b$
4	a <= b	*a*.is_leq(*b*)	Vergleich $a \leqslant b$
4	$a > b$	*a*.is_gt(*b*)	Vergleich $a > b$
4	a >= b	*a*.is_geq(*b*)	Vergleich $a \geqslant b$
5	$a + b$	*a*.plus(*b*)	Addition $a + b$
5	$a - b$	*a*.minus(*b*)	Subtraktion $a - b$
6	$a * b$	*a*.times(*b*)	Multiplikation $a \times b$
6	a/b	*a*.quotient(*b*)	Division a/b
6	a div b	*a*.divide(*b*)	ganzzahlige Division a div b
6	a mod b	*a*.modulo(*b*)	ganzzahliger Rest a mod b
7	a ^ b	*a*.pow(*b*)	Potenzieren a^b
8	$-a$	*a*.minus	unäres Minus $-a$
8	not a	*a*.negate	Negation $\neg a$

Die Addition bindet also schwächer als die Multiplikation, usw.: $a + b * c = a + (b * c)$, $-1 \wedge i = (-1) \wedge i$.

Tabellen wie 8.2 gibt es für alle imperativen Programmiersprachen. Wesentliche Unterschiede finden sich häufig bei der Einordnung der booleschen Operationen und des unären Minus. So ist

in SATHER $a < b$ and $b < c$ erlaubt, während man in PASCAL $(a < b)$ and $(b < c)$ schreiben muß, da dort and Vorrang vor Vergleichen hat und daher $a < b$ and $b < c$ als $a < (b \text{ and } b) < c$ geklammert würde.

Die dritte Spalte interessiert erst in Kap. 10. In manchen Sprachen kann man die Bedeutung der Operatoren umdefinieren oder für zusätzliche Operandentypen erklären; die vierte Spalte gibt die Bedeutung im *Regelfall* an.

Addition und Multiplikation sind kommutativ. Wie bereits in Anhang B.2 ausgeführt, können wir nicht mit der Gültigkeit des Assoziativ- und Distributivgesetzes für Zahlen rechnen. In allen Sprachen wird für die 4 Grundrechenarten Linksassoziativität unterstellt, also $a+b+c = (a+b)+c$, $a-b-c = (a-b)-c$, $a*b*c = (a*b)*c$, $a/b/c = (a/b)/c$. SATHER unterstellt für das Potenzieren Rechtsassoziativität, $a \wedge b \wedge c = a \wedge (b \wedge c)$; viele andere Sprachen betrachten auch das Potenzieren als linksassoziativ, also $a \wedge b \wedge c = (a^b)^c = a^{b*c}$.

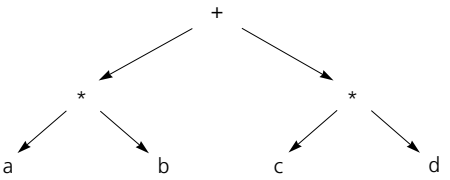

Abbildung 8.1: Kantorowitsch-Baum für $a * b + c * d$

Mit diesen Regeln können wir Ausdrücke in Kantorowitsch-Bäume überführen, wie dies Abb. 8.1 zeigt. Das Programm aus Beispiel 6.1 berechnet dann den Wert des Ausdrucks aus der zugehörigen Postfixform. In imperativer Notation lautet die Berechnung

```
h1  := a*b;                1
h2  := c*d;                2
res := h1 + h2             3
```

Allerdings ist dies nicht die einzige mögliche Reihenfolge der Berechnung von $a*b+c*d$: Auch die Reihenfolge 2, 1, 3 ist mit der Struktur des Kantorowitsch-Baums verträglich. Ebenso wäre parallele Berechnung von $a*b$ und $c*d$ erlaubt. Insbesondere kann man $a*b+a*b$ durch

```
h   := a*b;
res := h + h
```

berechnen.

Diese Bemerkungen gelten nicht nur für imperative, sondern auch für funktionale Sprachen. Im funktionalen Fall lassen sich die verschiedenen Reihenfolgen nicht unterscheiden: a, b, c, d sind zuvor berechnete, feste Werte. Im imperativen Programmieren bezeichnen a, b, c, d jedoch Zugriffswege zu Werten. Insbesondere könnten sich dahinter (parameterlose) Funktionsaufrufe verbergen, die durch

Veränderung des Zustands Nebenwirkungen auf den Wert des Ausdrucks haben, z. B. liefert

```
i: INT := 1;
a: INT is i := i+1; res := i end;
. . .
a*i + a*i
```

$2 * 2 + 3 * 3$, jedoch $2 * 2 + 2 * 2$, wenn wir $a * i$ nur einmal berechnen. Bestimmen wir den Wert von i, bevor wir die Prozedur a aufrufen, so könnte sich auch $1 * 2 + 2 * 3$ oder $1 * 2 + 1 * 2$ ergeben.

Aufgabe 8.2: Welche Ergebnisse könnte $a + i$ unter diesen Bedingungen liefern?

Das Ergebnis eines Ausdrucks oder, allgemeiner, eines Programmstücks heißt **undefiniert**, wenn verschiedene Implementierungen unterschiedliche Ergebnisse liefern könnten. Die Literatur spricht oft auch dann von einem undefinierten Ergebnis, wenn regelmäßig das gleiche, wenn auch unbekannte Ergebnis anfällt.

In den meisten imperativen Sprachen ist jede Reihenfolge der Bestimmung von Operanden und Ausführung von Operationen, die sich mit der durch den Kantorowitsch-Baum gegebenen Ordnung verträgt, auch zulässig. In der Terminologie von Abschnitt 1.4 können also die Werte der Operanden und die Ergebnisse getrennter Teilbäume zeitlich verzahnt oder kollateral bestimmt werden. Verändert die Bestimmung eines Operanden a den Wert eines anderen Operanden b, so sagen wir, die Berechnung von a habe eine **Nebenwirkung** (auf die Berechnung von b). Der Wert des Ausdrucks ist dann undefiniert.

Daß beliebige Reihenfolgen erlaubt sind, bedeutet, daß Nebenwirkungen verboten und folglich die obigen Ausdrücke $a * i + a * i$ usw. unzulässig sind. Im allgemeinen Fall ist die Frage, ob es Nebenwirkungen gibt, unentscheidbar. Ein Programmierer kann nicht erwarten, daß er vom Rechner auf unzulässige Ausdrücke hingewiesen wird.

Java und C# beseitigen dieses Problem, indem sie die Reihenfolge der Operandenzugriffe, wie sie sich aus der Postfixform ergibt, für verbindlich erklären. Jedoch bleibt auch dann ein Ausdruck, der Nebenwirkungen enthält, für den menschlichen Leser schwer verständlich und fehleranfällig. Das Gebot „Du sollst keine unverständlichen Programme schreiben“ verbietet Ausdrücke mit Nebenwirkungen aus nicht-technischen Gründen.

8.1.6 Ablaufsteuerung

Im zustandsorientierten Rechenmodell verändern Zuweisungen, sowie der Beginn und das Ende der Lebensdauer von Größen und Objekten den Zustand. Den Gesamtablauf, eines Programms steuern wir durch sequentielles, kollaterales oder paralleles Zusammensetzen solcher Zustandsübergänge. Wie bei den Formulierungen im Kochbuch in Abschnitt 1.4 gibt es dazu in Programmiersprachen neben Vereinbarungen und Zuweisungen noch Anweisungen zur bedingten oder

wiederholten Ausführung von (Teil-)Anweisungen. Ferner können wir mehrere Anweisungen zu einer Prozedur zusammenfassen.

Wir führen in diesem Abschnitt die Anweisungen für die **Ablaufsteuerung**[17] an Beispielen ein. Die genaue Schreibweise in SATHER ergibt sich aus den Syntax-Diagrammen in Anhang C.1. Wir verzichten vorläufig auf das parallele Zusammensetzen von Anweisungen.

8.1.6.1 Hintereinanderausführung, Blöcke

Das Hintereinanderausführen von Anweisungen $A_1, A_2, A_3, \ldots$ beschreiben wir durch eine **Anweisungsfolge** $A_1; A_2; A_3; \ldots$ mit Strichpunkt als Trennzeichen zwischen den Anweisungen. Wir hatten diese Schreibweise bereits in Absch. 8.1.5 benutzt. Der Strichpunkt ist sozusagen der *Sequentierungsoperator*.

Bewirkt eine Anweisung A_i einen Zustandsübergang $z_{i-1} \rightarrow z_i$, so liefert die Anweisungsfolge $A_1; A_2$ den **zusammengesetzten Zustandsübergang** $z_0 \rightarrow z_2$ mit Zwischenzustand z_1. Von außen betrachtet, interessieren nur die Zustände z_0, z_2. Der Zustand z_1 bleibt unsichtbar.

In Zwischenzuständen könnten nicht nur Variable andere Werte haben. Temporär könnte der Zustandsraum auch um zusätzliche Größen erweitert worden sein. Wir sehen das an der Vertauschung der Werte der Variablen i und j:

```
begin h: INT; h := i; i := j; j := h end
```

Der Zustandsraum besteht am Anfang und am Ende aus den beiden Variablen i und j. Zur Vertauschung benötigen wir temporär eine Hilfsvariable h, die vorher und nachher uninteressant ist. Dazu stellen wir der Anweisungsfolge die benötigte Vereinbarung für h voran und erhalten die Übergänge der Abb. 8.2.

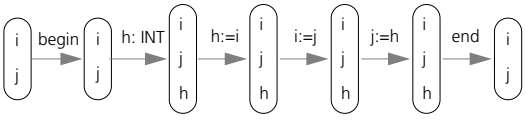

Abbildung 8.2: Änderungen des Zustandsraums beim Vertauschen von Werten

Eine Anweisungsfolge, der wie im Beispiel Vereinbarungen vorangestellt sein können, ist ein Block. Der Gültigkeitsbereich der in einem Block vereinbarten Bezeichner ist nach den Regeln aus Abschnitt 8.1.3 auf den Block beschränkt. Die Lebensdauer der vereinbarten Größen beginnt mit der Ausführung der Vereinbarung und endigt mit dem Ende der Ausführung des Blocks.

17. Der Ablauf eines Programms oder eines Teils davon heißt engl. *flow (of control)*. Dies veranlaßt manche, statt von Ablaufsteuerung von Ablaufkontrolle zu sprechen. Allerdings bedeutet im Deutschen „Kontrolle“ *Prüfung* oder *Überwachung* und nicht *Steuerung*.

Die Wortsymbole begin, end benutzen wir, um einen Block explizit zu kennzeichnen. begin kann als eine Leeranweisung aufgefaßt werden, die den Zustand nicht verändert. end verändert den Zustand; lokal vereinbarte Größen des Blocks werden ungültig.

Die **Leeranweisung** benötigen wir auch in anderem Zusammenhang. In theoretischen Erörterungen schreiben wir dafür leer. Im Programmtext steht dafür die leere Zeichenreihe.

8.1.6.2 Bedingte Anweisungen, Fallunterscheidungen

Aus funktionalen Sprachen sind uns bedingte Ausdrücke bekannt. In imperativen Sprachen gibt es eine entsprechende **bedingte Anweisung**, in SATHER etwa

```
if i > j then max := i else max := j end
```

Abhängig von der Bedingung i > j, im allgemeinen Fall ein beliebiger boolescher Ausdruck, wird die Ja- oder Nein-Alternative ausgewählt und ausgeführt. Beide Alternativen können Blöcke sein; begin und end sind nicht erforderlich.

In HASKELL schreiben wir für das Beispiel

```
max i j | i>j       = i
        | otherwise = j
```

oder `max i j = if i>j then i else j`

Das Beispiel

```
if i > j then h : INT; h := i; i := j; j := h end
```
(8.12)

zeigt eine **einseitige bedingte Anweisung**: Die Nein-Alternative ist eine Leeranweisung und wird samt dem Wortsymbol else weggelassen. Die bedingte Anweisung vertauscht die Werte der Variablen i und j, wenn vor der Ausführung $i > j$ gilt. Anschließend gilt immer $i \leqslant j$.

Beispiel 8.2: Die bedingte Anweisung

```
if i > j then p := true else p := false end
```

kann zur Zuweisung `p := i > j` vereinfacht werden. ♦

Programme werden sehr schwer verständlich, wenn die booleschen Ausdrücke zur Steuerung bedingter Anweisungen und Schleifen Nebenwirkungen haben. Wir setzen im folgenden stets voraus, daß solche Nebenwirkungen nicht vorliegen, selbst, wenn wir das nicht explizit erwähnen.

Häufig benötigen wir Kaskaden

```
if ...
then ...
else if ...
   then ...
   else ...
   end
end
```

in denen die Nein-Alternative selbst wieder eine bedingte Anweisung ist. Diese Konstruktion kürzen wir in SATHER (und ähnlich in anderen Sprachen) zu

```
if ...
then ...
elsif ...
then ...
else ...
end
```

Wenn eine Nein-Alternative nur aus einer einzigen bedingten Anweisung besteht, ziehen wir also **else if** zu **elsif** zusammen und lassen ein **end** weg.

Beispiel 8.3: *a* und *signum* seien Variable vom Typ **INT**. Die bedingte Anweisung

```
if a > 0 then signum := 1
elsif a < 0 then signum := -1
else signum := 0
end
```

berechnet in *signum* das Vorzeichen von *a*. ♦

Beispiel 8.4 (**H. D. DEMUTH, 1956**): Wir wollen die Werte von 5 Variablen a, b, c, d, e so vertauschen, daß $a \leqslant b \leqslant c \leqslant d \leqslant e$ gilt. Nach Abschnitt 5.3.4.1 bedeutet das, daß wir a, b, c, d, e **sortieren**. Wir sortieren zunächst die Paare (a, b) und (c, d) und dann die beiden größten Elemente, also (b, d). Dazu setzen wir jeweils die bedingte Anweisung (8.12) ein. Dies liefert eine der Konfigurationen der Abb. 8.3. Die Pfeile geben die bereits bekannten Größenbeziehungen an. Danach setzen wir mit Hilfe zweier Vergleiche das Element *e* in die Kette $[a, b, d]$

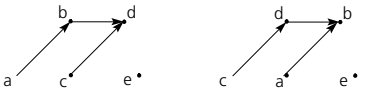

Abbildung 8.3: Sortieren von 5 Zahlen: Nach den Vergleichen $a > b$, $c > d$, $b > d$, vor eventuellem Vertauschen von b, d

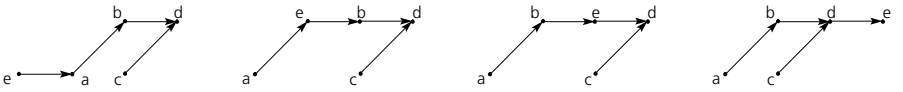

Abbildung 8.4: Sortieren von 5 Zahlen: Nach Vergleich mit *e*

ein. Die Vergleiche liefern eine der Konfigurationen der Abb. 8.4. Die ersten drei Konfigurationen der Abb. 8.4 geben die schwächste Kenntnis wieder, die wir besitzen. Diese nutzen wir, um mit zwei weiteren Vergleichen das Element *c* in die Viererkette einzusetzen und erhalten das Programm

```
if a>b then h: INT; h := a; a := b; b := h end;                                  1
if c>d then h: INT; h := c; c := d; d := h end;                                  2
if b>d then h: INT; h := b; b := d; d := h; h := a; a := c; c := h end;          3
if b>e                                                                           4
then if a>e then h: INT; h := e; e := d; d := b; b := a; a := h                  5
     else h: INT; h := e; e := d; d := b; b := h                                 6
     end                                                                         7
elsif d>e then h: INT; h := e; e := d; d := h                                    8
end;                                                                             9
if c>b                                                                           10
then if c>d then h: INT; h := c: c:= d; d := h end                               11
elsif c>a then h: INT; h := b; b := c; c := h                                    12
else h: INT; h := a; a:= c; c := b; b := h                                       13
end                                                                              14
```

Insgesamt führen wir genau 7 Vergleiche durch.

Dieses Programmstück läßt sich noch verbessern. Zum einen ist es nicht nötig, die in allen Alternativen vorkommende Hilfsvariable *h* jedes Mal neu zu vereinbaren. Wir könnten stattdessen das ganze Programmstück zu einem Block machen, in dem wir *h* einmal einführen:

```
begin
  h: INT;
  if a>b then h := a; a := b; b := h end;
  -- wie bisher, aber ohne die Vereinbarungen von h
end
```

Zum anderen haben wir nicht alle Kenntnisse genutzt, die wir laut Abb. 8.3 und 8.4 besitzen. Auf weitere Umformulierungen kommen wir auf S. 25 zurück. ♦

Aufgabe 8.3: Zeigen Sie, daß man zum Sortieren von 5 beliebigen Zahlen mindestens 7 Vergleiche benötigt.

Aufgabe 8.4: Wieviele Vergleiche braucht man mindestens, um 6 Zahlen zu sortieren? Geben Sie hierfür ein Programmstück an.

Auf S. 47 werden wir sehen, wie wir uns von der Richtigkeit unseres Programms überzeugen können. Wollen wir das durch Testen erreichen, so müßten wir prüfen, ob unser Programm sämtliche $5! = 120$ Permutationen fünf verschiedener Zahlen richtig sortiert. Das erschöpfende Testen von if-then-else-Programmen kann also sehr schnell zu einer großen und nicht mehr beherrschbaren Anzahl von Testfällen führen. Im vorliegenden Fall genügen allerdings bereits $2^5 = 32$ Testfälle:

Aufgabe 8.5: Zeigen Sie: Ein Sortierprogramm zum Sortieren von n Zahlen, das sich aus bedingten Anweisungen der Form (8.12), also aus Vergleichen mit anschließendem Vertauschen, zusammensetzen läßt, ist bereits dann richtig, wenn es beschränkt auf Zahlen mit den Werten 0 und 1 richtig arbeitet.

Aufgabe 8.6: Würde die zusätzliche Berücksichtigung der weiteren Konfigurationen aus Abb. 8.3 und 8.4 tatsächlich den mittleren Aufwand des Sortierens von 5 Zahlen erheblich senken? Oder nur den Programmieraufwand (und die Fehleranfälligkeit) erhöhen? Nehmen Sie dazu an, daß die 120 Permutationen gleichverteilt als Eingabe auftreten könnten. Ermitteln Sie, wie oft die einzelnen Konfigurationen vorkommen, und wieviele Vergleiche und Zuweisungen Sie jeweils einsparen könnten.

Eine solche Analyse sollte man auch bei komplexeren Aufgaben ausführen, *bevor* man sich mit der Umformulierung des Programms beschäftigt.

Aufgabe 8.7: Geben Sie zu Beispiel 8.4 ein geordnetes binäres Entscheidungsdiagramm (OBDD, vgl. 4.1.8) an, das den Entscheidungsprozeß wiedergibt, und überzeugen Sie sich damit von der Richtigkeit des Programms. Anleitung: Jede Permutation ist bekanntlich durch die Inversionen ihrer Elemente charakterisiert. Ein Vergleich wie $c < d$ prüft, ob eine Inversion der beiden Elemente vorliegt oder nicht. Die vorhandenen Inversionen definieren zu Beginn die booleschen Werte für unser OBDD. Aufgrund der Vertauschungen bezeichnet allerdings $c < d$ nicht immer die gleiche Inversion.

Oft haben wir es mit geschachtelten bedingten Anweisungen

```
if i = k1 then B1
elsif i = k2 then B2
elsif i = k3 then B3
...
else B0
end
```

zu tun, in denen der Wert eines Ausdrucks i mit verschiedenen Werten k_1, k_2, usw. verglichen und dann ein Block B_i ausgeführt wird.
Diese können wir kürzer als Fallunterscheidung

```
case i
when k1 then B1
when k2 then B2
when k3 then B3
...
else B0
end
```

schreiben, in der der Ausdruck i nur ein einziges Mal berechnet wird. Die k_j müssen allerdings Literale der Typen INT, BOOL oder CHAR sein. Wir werden in Abschnitt 11.3.2 sehen, wie sich die Auswahl unter n Fällen mit Aufwand O(1) bewerkstelligen läßt, wenn die **Fallmarken** k_j eine (nahezu) lückenlose Folge bilden. Lückenlosigkeit ist allerdings keine Voraussetzung für den Einsatz der Fallunterscheidung; auch können die Fälle in beliebiger Reihenfolge angegeben sein; schließlich kann man mehrere Fallmarken k_j, $k_{j'}$ durch Komma getrennt als

Liste when k_j, $k_{j'}$ then ... angeben, wenn die Blöcke B_j und $B_{j'}$ identisch sind. Sämtliche Fallmarken k_j müssen verschieden sein.

Die Nein-Alternative else B_0 einer Fallunterscheidung wird gewählt, wenn keiner der explizit genannten Fälle vorliegt; sie kann auch fehlen. Das Fehlen der Nein-Alternative kann zu ähnlichen Problemen führen, wie wir sie im funktionalen Programmieren kennenlernten, wenn die Alternative otherwise nicht angegeben war.

8.1.6.3 Schleifen

Schleifen hatten wir in Abschnitt 5.5.1 als den Spezialfall der Rekursion kennengelernt, der in der Mathematik dem Induktionsbeweis entspricht. Mit einer Schleife können wir den Wert von Zustandsvariablen solange modifizieren, bis eine bestimmte Zielbedingung erfüllt ist. Im zustandsorientierten Programmieren sind Schleifen das mächtigste Instrument des Programmierens. Zusammen mit der Zuweisung und dem Hintereinanderausführen von Anweisungen gestattet es beliebige Programme zu formulieren. Wir werden dies in Bd. III beweisen.

Die Standardform der Schleife ist in den meisten imperativen Sprachen die **while-Schleife**

while *Bedingung* loop *Block* end

mit einem booleschen Ausdruck als Schleifenbedingung. Sie entspricht dem Funktional while aus Abschnitt 5.5.1. Von dort übernehmen wir auch die Begriffe **Schleifenrumpf** für den Block und **Schleifeninvariante** für ein Prädikat P, das vor und nach der Ausführung des Schleifenrumpfes auf den Zustand zutrifft.

Der Schleifenrumpf wird ausgeführt, solange die Schleifenbedingung wahr ist. Die Bedeutung der Schleife können wir daher rekursiv erklären durch

if *Bedingung* then *Block*; while *Bedingung* loop *Block* end end (8.13)

Schleifen könnten endlos laufen. Wie in Abschnitt 5.5.1 unterscheiden wir zwischen der **totalen** und der **partiellen Korrektheit** eines Programms. Zum Nachweis totaler Korrektheit benötigen wir eine **Terminierungsfunktion** $t(x)$, die vom Zustand x abhängt und bei vorgegebenem Anfangszustand nur endlich viele Werte annehmen darf. Wenn wir nachweisen können, daß die Schleife nur endlichen Aufwand verursacht, können wir uns die Terminierungsfunktion sparen. Endlicher Aufwand zusammen mit partieller Korrektheit garantiert auch totale Korrektheit.

Beispiel 8.5: Beispiel 5.41 berechnete den größten gemeinsamen Teiler zweier nicht-negativer ganzer Zahlen in Haskell durch

```
ggT a b = fst (until p gg' (a,b))
          where p (x,y)  =  y==0
                gg' (x,y) = (y,x `rem` y)
```

Mit einer while-Schleife erhalten wir

```
while b /= 0 loop h: INT := a; a := b; b := h mod b end;
-- Resultat in a
```

Das Ergebnis erhalten wir durch wiederholte Modifikation der Werte von a und b unter Einsatz der Hilfsvariablen h. Die while-Schleife ersetzt den Aufruf von until aus dem HASKELL-Programm. Der Schleifenrumpf entspricht der Funktion gg', die Schleifenbedingung der Funktion p. ♦

Beispiel 8.6: In Aufgabe 5.43 berechneten wir das kleinste gemeinsame Vielfache kgV(a, b, c) dreier ganzer Zahlen $a, b, c > 0$. In SATHER lautet dieses Programm:

```
A,B,C: INT;
A := a; B := b C ;= c;
while A /= B or B /= C loop
    if    A<B  then A := A+a
    elsif B<C  then B := B+b
    else            C := C+c -- Fall C<A
    end
end;
-- Ergebnis in A, B und C
```

Eine mögliche Schleifeninvariante ist: A, B, C sind Vielfache von a, b, c und $A, B, C \leqslant \text{kgV}(a, b, c)$, oder als prädikatenlogische Formel:

$$\begin{aligned} \exists p, q, r: \quad & A = p\,a \wedge B = q\,b \wedge C = r\,c \wedge \\ & A \leqslant \text{kgV}(a, b, c) \wedge \\ & B \leqslant \text{kgV}(a, b, c) \wedge \\ & C \leqslant \text{kgV}(a, b, c). \end{aligned}$$

Bei Schleifenabbruch gilt

$$p\,a = q\,b = r\,c \leqslant \text{kgV}(a, b, c).$$

Dies ist nur möglich, wenn $A = B = C = \text{kgV}(a, b, c)$. Da sich in jedem Schleifendurchlauf entweder A, B oder C erhöht und $\text{kgV} \leqslant a \cdot b \cdot c$ gilt, fällt die Funktion $t(A, B, C) = 3 \cdot a \cdot b \cdot c - A - B - C$ streng monoton und kann keine negativen Werte annehmen. Sie eignet sich deshalb als Terminierungsfunktion unserer Schleife; diese bricht also ab. Das Programmstück ist total korrekt. ♦

Die meisten imperativen Sprachen kennen neben der while-Schleife auch eine **Zählschleife**. Man benutzt sie etwa, um die Elemente einer Reihung zu durchlaufen. Die Zählschleife weist einem Zähler einen Anfangswert zu. Solange der Endwert nicht erreicht ist, wird der Schleifenrumpf ausgeführt und der Zähler um die Schrittweite erhöht oder erniedrigt. Sofern der Schleifenrumpf

terminiert und den Zähler nicht manipuliert, terminiert auch die Zählschleife als Ganzes.

In SATHER notiert man Zählschleifen mit den Zählungen *anfang*.upto!(*ende*) bzw. *anfang*.downto!(*ende*). Beide Konstruktionen sind Spezialfälle von Strömen, auf die wir in Abschnitt 10.4.2 genauer eingehen. Sie erhöhen bzw. erniedrigen einen Zähler jeweils um 1, bis der Endwert *ende* erreicht ist; die Anzahl der Schleifendurchläufe ist daher $|ende - anfang| + 1$. Bei der Zählschleife kann die while-Bedingung entfallen:

```
loop zähler := anfang . upto! (ende); Anweisung end
```

Die äquivalente Schreibweise als while-Schleife lautet:

```
zähler := anfang;
while zähler <= ende
loop Anweisung;
   zähler := zähler + 1
end
```

Eine Zählschleife in PASCAL hat die Form for *zähler* := *anfang* to *ende* do *Anweisungsfolge* bzw. for *zähler* := *anfang* downto *ende* do *Anweisungsfolge*. Wie in SATHER ergeben sich $|ende - anfang| + 1$ Durchläufe. Zusätzlich kennt PASCAL eine weitere Spielart der while-Schleife: Die **until-Schleife** repeat *Anweisungsfolge* until *Bedingung* führt den Schleifenrumpf mindestens einmal aus und terminiert, sobald die Bedingung erfüllt ist. Sie ist gleichwertig zu *Anweisungsfolge;* while not *Bedingung* do begin *Anweisungsfolge* end.

C, C++ und JAVA kennen keine Zählschleifen. Ihre **for-Schleife** schreibt sich for(*Init* ; *Bedingung* ; *Ende*) *Anweisung* und entspricht der while-Schleife *Init* while(*Bedingung*) { *Anweisung* ; *Ende* }. Anstelle der until-Schleife bieten diese Sprachen die **do-while-Schleife**. Sie hat die Form do *Anweisung* while (*Bedingung*) und ist äquivalent zu *Anweisung* ; while(*Bedingung*) *Anweisung*. In C# können Zählschleifen ähnlich wie in SATHER definiert werden:

```
IEnumerate<int> strom = FromTo(anfang,ende);
foreach (int zaehler in strom) Anweisung;
```

Außerdem kennt C# auch eine for-Schleife wie in C, C++ und JAVA.

Beispiel 8.7: Gegeben sei eine Reihung *a*: ARR[n](INT). Wir wollen die Elemente zusammenzählen.
Dazu schreiben wir

```
s: INT := 0;
loop constant i: INT := 0.upto!(n-1); s := s + a[i] end
```

Das innere Produkt $\sum_{i=0}^{n-1} a_i b_i$ zweier Vektoren *a*, *b*: ARR[n](FLT) erhalten wir durch

```
s: FLT := 0.0;
loop constant i: INT := 0.upto!(n-1); s := s + a[i]*b[i] end
```

Als Zähler eignet sich jede beliebige Größe vom Typ INT; sie muß nicht notwendig lokal in der Schleife vereinbart sein. Daß wir in diesem und allen

folgenden Beispielen den Zähler als lokale Größe der Schleife einführen und noch dazu zur Konstanten erklären, ist eine Sicherheitsmaßnahme, die uns vor unbeabsichtigten Fehlern schützen soll: Wir vermeiden so mit Sicherheit, daß eine geschachtelte innere Schleife einen Zähler „aus Versehen" nochmals benutzt, oder daß, etwa auch durch einen Schreibfehler, der Wert des Zählers durch eine Zuweisung verändert wird.

In der Sprache ADA sind Zähler immer Konstante; es gibt keine Alternative. In PASCAL und MODULA-2 sind Zähler hingegen immer Variable. Vorbeugemaßnahmen wie das Konstant-Setzen von Zählern zum Schutz gegen Fehler sind im Jargon als **defensives Programmieren** bekannt. ♦

Beispiel 8.8: Den FLOYD-WARSHALL-Algorithmus zur Berechnung der transitiven Hülle einer Relation ρ, Programm 2.1 aus Bd. I, zeigt Programm 8.1 in SATHER.

Programm 8.1: FLOYD-WARSHALL-Algorithmus

```
-- A : ARR[n, n](INT) sei vorgegebene Adjazenzmatrix
S: ARR[n,n](INT) := A;
loop constant i: INT := 0.upto!(n-1); S[i,i] := 1 end;
loop constant i: INT := 0.upto!(n-1);
  loop constant k: INT := 0.upto!(n-1);
    loop constant j: INT := 0.upto!(n-1); S[i,j] := S[i,j] + S[i,k]*S[k,j]
    end
  end
end
-- S ist jetzt die Adjazenzmatrix der reflexiven, transitiven Hülle von A
```

Die Gesamtzuweisung $S := A$ zur Vorbesetzung der Reihung S in diesem Beispiel muß in vielen Programmiersprachen, z. B. in PASCAL und C, in Form einer Doppelschleife mit Zuweisungen $S[i,j] := A[i,j]$ einzeln ausgeschrieben werden. ♦

Schleifen aller Arten können in SATHER auch explizit durch die **Abbruchanweisung** break abgebrochen werden. Die Schleifen

loop Anweisung end

und

loop if not (Bedingung) then break end; Anweisung end

sind äquivalent.

8.1.6.4 Prozeduren

Ein Block abstrahiert von den Einzelheiten, mit denen ein Zustandsübergang erreicht wird: Von außen betrachtet bildet er eine einzelne Anweisung, die mehrere Variable zugleich ändern kann: wie und mit welchen lokalen (Hilfs)-Variablen dies erreicht wird, ist von außen nicht sichtbar. Es handelt sich um einen zusammengesetzten Zustandsübergang.

Wenn der gleiche Block B in einem Programm mehrfach benötigt wird, geben wir ihm einen Namen p und schreiben die in Abschnitt 8.1.2 eingeführte Vereinbarung

procedure p is B end

einer echten Prozedur. p kann nach dieser Vereinbarung anstelle von B wie eine Anweisung benutzt werden. Diese Verwendung von p heißt ein **Prozeduraufruf**. Wenn eine Prozedur f ein Ergebnis liefern soll, das als Wert vom Ergebnistyp T in einen Ausdruck eingeht, so heißt f wie in Abschnitt 8.1.2 beschrieben eine **Funktion(sprozedur)** und wird mit

procedure f : T is B end

vereinbart.

Aufgabe 8.8: Erklären Sie, warum es in HASKELL keine echten Prozeduren, sondern nur Funktionsprozeduren gibt.

Wie in Abschnitt 8.1.3 bemerkt, wird in SATHER die Ergebnisvariable res in Funktionsprozeduren implizit mit dem Ergebnistyp, hier INT, vereinbart.

Zur Kennzeichnung des Funktionsergebnisses sind verschiedene Verfahren üblich: In ALGOL 60 und PASCAL muß der Rumpf einer Funktion f (mindestens) eine Zuweisung an eine implizit vereinbarte Variable f : T des gleichen Namens wie die Funktion ausführen. In MODULA-2 C, C++, JAVA und C# schreibt man return Ausdruck, um das (dynamische) Ende der Ausführung der Funktion und gleichzeitig den Wert des Ausdrucks als Funktionsergebnis zu kennzeichnen. Zusätzlich darf man in SATHER wie in C, C++, JAVA und C# die Anweisung **return** in einer Methode oder Funktion schreiben, um die Ausführung der Methode zu beenden.

Prozeduren dienen in imperativen Sprachen zwei Aufgaben:

- Abstraktion von den Einzelheiten eines zusammengesetzten Zustandsübergangs;
- Zusammenfassung eines mehrfach vorkommenden Blocks zu einer (benannten) Einheit;

Beide Effekte treten meistens zusammen auf. Wir sprechen dann auch von **prozeduraler Abstraktion** und nennen eine Prozedur häufig eine (zusammengesetzte) Operation, da wir sie wie in funktionalen Sprachen zur Realisierung von Operationen auf Datenstrukturen einsetzen.

Prozeduren können Parameter haben, für die beim Aufruf Argumente eingesetzt werden. Für die Parameter muß ein Name und ein Typ spezifiziert sein:

```
procedure max(i,j: INT): INT is if i<j then res := j else res := i end end
```

Man kann dann die Funktion z. B. mit max(k − 1, 3) aufrufen; als Argumente sind beliebige Ausdrücke entsprechenden Typs zulässig.

In funktionalen Sprachen ersetzen die Argumente beim Aufruf einer Prozedur die Parameter. Beim zustandsorientierten Programmieren mit imperativen Sprachen unterscheiden wir verschiedene Arten von Parametern und unterschiedliche Mechanismen zur **Parameterübergabe**:

- **Eingabeparameter**: Die Prozedur ist eine Rechenvorschrift, die auf verschiedene Argumente angewandt werden kann. Eingabeparameter bringen diese Argumente in die Prozedur ein. Dies ist die häufigste Art von Parametern. Die Parameter mathematischer Funktionen, von Funktionen in HASKELL und auch die Parameter der vorangehend definierten Prozedur max sind Eingabeparameter.

  Zur Parameterübergabe benutzen wir **Wertaufruf**[18]: Die Spezifikation des Parameters wird als lokale Vereinbarung einer Variablen im Prozedurrumpf aufgefaßt. Beim Aufruf der Prozedur wird diese Variable mit dem Wert des Arguments vorbesetzt. Darf der Wert dieser Variablen in der Prozedur anschließend nicht mehr verändert werden, so spricht man von **striktem Wertaufruf**.
- **Ausgabe**- oder **Ergebnisparameter**: Programm 8.2 berechnet aus Eingabeparametern a, b den $\mathrm{ggT}(a, b)$ und zusätzlich die Werte u, v mit $a * u + b * v = \mathrm{ggT}(a, b)$. Das Ergebnis besteht also aus mehreren Werten $u, v, \ldots$ In HASKELL definieren wir dazu eine Funktion eggT mit den Eingabeparametern a, b und dem Ergebnis (u, v, z) mit $z = \mathrm{ggT}(a, b)$. Diese Lösung ist auch in vielen imperativen Sprachen möglich. Sie ist aber eigentlich unerwünscht, da hier oft logisch nicht zusammengehörige Dinge zu einer Einheit zusammengeschlossen werden. Ergebnisparameter a lösen dieses Problem, indem sie am Ende der Ausführung einer Prozedur ihren Wert durch eine Zuweisung $x := a$ an den Aufrufer abliefern. x ist das Argument, das wir beim Aufruf für den Ergebnisparameter übergeben. Offensichtlich muß es sich um eine Variable handeln, an die man zuweisen kann.

  Zur Parameterübergabe benutzen wir **Ergebnisaufruf**[19]: Wie bei Eingabeparametern wird aus der Parameterspezifikation eine lokale Variable des Prozedurrumpfs. Diese ist jedoch nicht vorbesetzt. Ihr Wert wird am Ende eines Prozeduraufrufs dem Argument zugewiesen. Zur Unterscheidung von Eingabeparametern müssen wir Ergebnisparameter speziell kennzeichnen. In SATHER wird der Spezifikation eines Ergebnisparameters und seinem Argument beim Aufruf ein Kaufmanns-Und & vorangestellt. Als Ergebnis eines Aufrufs $z :=$ eggT$(40902, 24140, \&x, \&y)$ des Programms 8.2 erhalten wir $z = 34, x = 337, y = -571$, da $34 = 337 \cdot 40902 - 571 \cdot 24140$ gilt.

18. engl. *call by value*.
19. engl. *call by result*.

Programm 8.2: Berechnung von u, v mit $a * u + b * v = \text{ggT}(a, b)$

```
eggT(a,b: INT; & u: INT; & v : INT): INT is
-- berechne ggT von a und b, sowie u und v,
-- mit ggT(a,b) = a*u + b*v
  q,u1,v2,h1,h2,h: INT;
  u := 1; v := 0;
  u1 := 0; v2 := 1;
  while b /= 0 loop
    q := a div b;
    h1 := u-u1*q; u := u1; u1 := h1;
    h2 := v-v2*q; v := v2; v2 := h2;
    h := a-b*q; a := b; b := h;
  end;
  res := a;
end;
```

- **Transienter Parameter**: Wenn eine Prozedur den Wert einer Zustandsvariablen z ändern soll, benötigt sie oft den bisherigen Wert von z als Eingabe und liefert dann den neuen Wert als Ausgabe zurück. Diese Kopplung von Eingabe- und Ausgabeparameter zu einem Parameter heißt transienter Parameter: Den Wert des Arguments übernehmen wir bei Aufruf der Prozedur; am Ende weisen wir das Ergebnis wieder an das Argument zu. Dieses muß also wie bei Ergebnisparametern eine Variable sein.

  Zur Parameterübergabe kann man **Wert-Ergebnisaufruf**[20], d. h. die Kopplung des Wert- und Ergebnisaufrufs benutzen. Die lokale Variable wird mit dem Argument vorbesetzt; ihr Wert wird am Ende wieder dem Argument zugewiesen. In SATHER charakterisieren wir den Wert-Ergebnisaufruf durch zwei Kaufmanns-Unds && vor der Parameterspezifikation und dem transienten Argument im Aufruf.

Viele andere Programmiersprachen benutzen für transiente Parameter den **Referenzaufruf**[21]. Der Prozedur teilt man die Referenz der Variablen v mit, die als transientes Argument benutzt wird. Wenn die Prozedur einen solchen Referenzparameter liest oder schreibt, wird auf den Wert der Variablen v zugegriffen. Insbesondere wirkt eine Zuweisung an den transienten Parameter stets sofort auf das Argument v zurück, nicht erst bei Prozedurende. In PASCAL oder MODULA-2 heißen transiente Parameter mit Referenzaufruf var-Parameter.

Technisch ist Referenzaufruf meist mit geringerem Aufwand verbunden als Wert-Ergebnisaufruf. Letzterer kopiert beispielsweise eine Reihung, die als transientes Argument benutzt wird, zweimal, je einmal zu Beginn und zu Ende der

20. engl. *call by value-result*.
21. engl. *call by reference*.

Prozedur; der Referenzaufruf übergibt stattdessen nur die Referenz der Reihungsvariablen. Bei Anwendungen in verteilten Systemen ist Referenzaufruf jedoch unmöglich, wenn Rechnergrenzen überschritten werden: Der Referenzaufruf muß dann mit Wert-Ergebnisaufruf simuliert werden.

Die Programmiersprachen FORTRAN und ADA schreiben bei Reihungen im Unterschied zu Parametern anderer Typen stets Referenzaufruf vor, um den Mehraufwand zu vermeiden.

Neben diesen Verfahren gibt es noch den weniger verbreiteten **Namensaufruf**, bei dem das Argument als Funktionsprozedur aufgefaßt wird, die bei jedem Zugriff auf den Parameter aufgerufen wird, sowie den **Makroaufruf**, bei dem der Text des Arguments anstelle des Parameters im Prozedurrumpf eingesetzt wird. Letzterer kann die Zuordnung der im Argument vorkommenden Bezeichner zu ihren Vereinbarungen in unerwünschter Weise ändern.

Beispiel 8.9: Die Unterschiede zwischen den einzelnen Verfahren der Parameterübergabe sehen wir an folgendem (abschreckenden) Beispiel, in dem ?? anzeigt, daß verschiedene Übergabeverfahren einzusetzen sind:

```
m: INT:=1; n: INT;
procedure p(?? j: INT; ?? k: INT): INT is
  j:=j+1; m:=m+k; res :=j+k;
end;
-- Aufruf:
n := f(m,m+3);
```

Die Ergebnisse zeigt Tab. 8.3. Wert-Ergebnisaufruf und Referenzaufruf sind für *k* unzulässig: als Argument muß in diesen Fällen eine Variable angegeben sein, nicht ein Ausdruck wie m + 3. Der Gebrauch einer Variablen als Argument für

Tabelle 8.3: Ergebnisse bei unterschiedlicher Parameterübergabe

Mechanismus	m	n	j	k	Kommentar
Wertaufruf	5	6	2	4	Strikter Wertaufruf wegen Zuweisung an *j* nicht möglich
Wert-Ergebnisaufruf	2	6	2	4	nur zulässig für *j*
Referenzaufruf	6	10	6	4	nur zulässig für *j*
Namensaufruf	7	17	7	10	

zwei verschiedene Parameter im Ergebnis- oder Referenzaufruf ist fehleranfällig. Auch die Verwendung von Argumenten, die gleichzeitig als globale Variable im Prozedurrumpf vorkommen, sollte man vermeiden. ♦

Aufgabe 8.9: Was liefert das Beispielprogramm bei Ergebnisaufruf?

Beispiel 8.10: Mit Wert-Ergebnisaufruf können wir das Vertauschen zweier Werte und die bedingte Anweisung (8.12) als Prozeduren formulieren:

```
tausche(&& i,j: INT) is h: INT := i; i := j; j := h end;
minmax(&& i,j: INT) is if i>j then tausche(&& i, && j) end
```

Da wir die beiden Parameter in einer Spezifikation angeben, brauchen wir && auch nur einmal schreiben; im Aufruf muß das doppelte && jedoch vor jedem Argument stehen.

Damit können wir das Beispiel 8.4 nun übersichtlich schreiben:

```
minmax(&& a, && b);
minmax(&& c, && d);
if b > d then tausche(&& b, && d); tausche(&& a, && c); end;
if b > e
then if a > e then tausche(&& e, && a); tausche(&& e, && b); tausche(&& e, && d);
    else tausche(&& e, && b); tausche(&& e, && d)
    end
else minmax(&& d, && e)
end;
if c>b
then minmax(&& c, && d)
elsif c>a then tausche(&& b, && c)
else tausche(&& a, && c); tausche(&& b, && c)
end
```

♦

Prozeduren und Funktionen können in imperativen Sprachen genauso wie in funktionalen Sprachen rekursiv benutzt werden. Rekursion setzen wir zu den gleichen Zwecken ein, die wir in Kap. 5 für funktionale Programme erörterten. Da wir über Schleifen verfügen, kommt Rekursion in imperativen Programmen bei weitem nicht so häufig vor. Vor allem bei Teile-und-Herrsche-Algorithmen ist sie jedoch auch im imperativen Programmieren unentbehrlich.

Von den heute verbreiteten Programmiersprachen erlauben COBOL und ältere Versionen von FORTRAN keine rekursiven Prozeduren.

8.1.6.5 Ausnahmebehandlung

Mißverständnisse bei der Abfassung der Aufgabenstellung, Fehlinterpretation der Dokumentation, Schreibfehler in der Eingabe, Ablenkung oder Übermüdung bei der Bedienung von Rechnern, Ressourcenbeschränkungen oder Hardwarefehler können auch bei an sich korrekter Software zu fehlerhaften und so nicht vorgesehenen Zuständen im Programmablauf führen. Theoretisch hilft dagegen ein beständiges, penibles Überprüfen aller Eingangsgrößen. Praktisch sind diesem Verfahren Grenzen gesetzt, weil sich Inkonsistenzen oft nur mit einem Aufwand ermitteln lassen, der in der gleichen Größenordnung oder sogar höher liegt als der Lösungsaufwand des Problems.

Beispiel 8.11: Die Lösung eines linearen Gleichungssystems $\mathfrak{A}\mathfrak{x} = \mathfrak{b}, \mathfrak{b} \neq \mathfrak{O}$ setzt voraus, daß die Matrix $\mathfrak{A}$ nicht singulär ist. Bei Anwendung des Gaußschen Eliminationsverfahrens muß man die Matrix auf Dreiecksform reduzieren, um dies zu prüfen. Damit ist aber bereits der größere Teil des Aufwands zur Lösung des Gleichungssystems geleistet.

Die Multiplikation zweier ganzer Zahlen i, j führt bei 32 Bit Arithmetik zum Überlauf, wenn $\operatorname{ld} i + \operatorname{ld} j \geqslant 31$ gilt. Es ist billiger, statt der Logarithmen versuchsweise $i * j$ zu berechnen. ♦

Um solche Situationen mit wirtschaftlich vertretbarem Aufwand abfangen zu können, sehen viele moderne Programmiersprachen, so auch SATHER, sogenannte **Ausnahmen**[22] vor. Eine Ausnahme ist ein Ereignis, durch das die normale Ausführungsreihenfolge unterbrochen wird, um den aufgetretenen Fehler zu behandeln. Jede Ausnahme hat einen (**Ausnahme**-)**Typ**, um verschiedene Fehlerursachen unterscheiden zu können. Der Ausnahmebehandlung wird ein Ausnahmeobjekt zur Verfügung gestellt, dem Einzelheiten der Fehlerursache, z. B. der Ort des Fehlers im Programm, entnommen werden können. Die Einzelheiten sind implementierungsabhängig.

In SATHER geben wir eine Ausnahmebehandlung wieder durch

```
begin Block except ausnahmebezeichner
        when Ausnahmetyp_1 then Block_1
        when Ausnahmetyp_2 then Block_2
        ...
        else Block_0
end
```

ausnahmebezeichner benennt das Ausnahmeobjekt. Wie in der Fallunterscheidung folgen when-Klauseln. Diesmal wird aber nicht nach Literalen unterschieden, die Wert des Ausnahmebezeichners sein könnten, sondern nach dem Typ der Ausnahme, z. B. INTEGER_OVERFLOW, ZERO_DIVIDE, usw. Der anschließende Block_i beschreibt die Ausnahmebehandlung. Die Nein-Alternative faßt die Ausnahmebehandlung aller nicht explizit zuvor genannten Fehler zusammen; sie könnte auch fehlen. Die normale Programmausführung wird nach dem end der gesamten Anweisung fortgesetzt.

Ziel der Ausnahmebehandlung ist es

1. die Fehlerursache zu ermitteln und eine Meldung auszugeben;
2. den Zustand so zu korrigieren, daß die Ausführung des Programms fortgesetzt werden kann;
3. einen geeigneten Aufsetzpunkt zu finden, an dem die normale Ausführung des Programms wieder aufgenommen wird.

Beispiel 8.12: Wir wollen die Tangens-Funktion im Intervall $[-\pi, \pi]$ tabulieren und in einer Reihung abspeichern. Hierzu sei eine Reihung a geeigneter Größe gegeben; ferner seien die Funktionen sin und cos zugänglich. Für Argumente wie $\pi/2$, an denen der Tangens unendlich wird, soll NaN abgespeichert werden.

22. engl. *exception*.

Dies leistet das Programmstück:

```
constant pi: FLT := 3.14159265358979323846;
constant schritt: FLT := pi/12.0;
argument: FLT := -pi;
loop constant i: INT := 0.upto!(24);
 begin
   a[i] := sin(argument)/cos(argument);
   except fehler
   when FLOAT_OVERFLOW then a[i] := NaN
 end;
 argument := argument + schritt
end
```

Wegen Rundungsfehlern werden die Argumentwerte $\pm\,\pi/2$ möglicherweise nicht exakt erreicht. Die Ausnahme tritt nicht zwingend auf. ♦

In unserem Beispiel ist Ziel 1 trivial, da die Ursache bekannt war. Wir hätten die Ausnahmebehandlung durch die bedingte Anweisung

```
if cos(argument)<1.0e-7 then a[i] := NaN
else a[i] :=  sin(argument)/cos(argument) end
```

ersetzen können.

Letzteres ist übrigens auch die Methode, mit der die Hardware unserer Rechner auf ihr bekannt werdende Ausnahmen – wir nennen sie **Alarme** – reagiert: Nach Ausführung jedes Befehls wird automatisch abgefragt, ob eine Ausnahmebedingung, z. B. ein Fehler in der Arithmetik, aufgetreten ist. Gegebenenfalls wird eine Ausnahmebehandlung eingeleitet.

Die Ausnahmebehandlung in C, Pascal und Modula-2 benutzt explizite Abfragen, um einen Fehler zu finden; falls er nicht korrigiert werden kann, wird er nach außen gemeldet, indem man statt des Standardergebnisses 0 eine (meist negative) Fehlernummer liefert. Der Aufrufer reagiert auf die Fehlernummer so, als ob der Fehler bei ihm selbst aufgetreten wäre. Dieses Verfahren wird auch im Betriebssystem Unix benutzt.

Die Ausnahmebehandlung in Ada, Java und C# folgt den gleichen Prinzipien wie in Sather. Zusätzlich gehören in Java und C# die möglicherweise von einer Prozedur ausgelösten Ausnahmen zur Schnittstelle der Prozedur und müssen im Prozedurkopf nach den Parametern und dem Ergebnistyp spezifiziert werden.

Punkt 2 und 3 sind im Beispiel ebenfalls trivial, da durch die Erhöhung von argument automatisch wieder ein brauchbarer Zustand hergestellt wird.

Ausnahmen werden entweder implizit durch die Hardware oder explizit durch eine Anweisung

```
raise Ausnahmetyp
```

im Programm ausgelöst. Innerhalb der Ausnahmebehandlung kann man durch raise ohne Angabe eines Ausnahmetyps die gleiche Ausnahme wieder auslösen. In diesem Fall gilt die Ausnahme als noch nicht (vollständig) behandelt.

In Sather wird die normale Ausführung des Programms am Ende des Blocks wieder aufgenommen, der die Ausnahmebehandlung enthielt. Enthält eine Prozedur keine Ausnahmebehandlung zu einer gegebenen Ausnahme (oder wurde

die Ausnahme nochmals ausgelöst), so wird der Prozeduraufruf automatisch beendigt, und beim Aufrufer nach einer Ausnahmebehandlung für den Fehler gesucht. In unserem Beispiel würde die Ausnahmebehandlung also auch reagieren, wenn innerhalb der sin- oder cos-Routine ein arithmetischer Überlauf auftreten würde, der dort nicht abgefangen wird. Enthält das gesamte Programm keine Ausnahmebehandlung, so wird der Programmlauf abgebrochen.

Bei Programmen, die ein bestimmtes Gesamtergebnis liefern sollen, ist ein solcher Abbruch oft unkritisch: Man nimmt zur Kenntnis, daß für den gegebenen Satz von Eingabedaten kein Ergebnis erzielt wurde. Bei Datenbankanwendungen oder bei der Steuerung eines Flugzeugs, allgemein bei reaktiven Systemen, müssen hingegen sorgfältig alle Fehler analysiert, ihre Ursachen aufgezeichnet und dann für Abhilfe gesorgt werden. Dazu muß bereits im Entwurf geklärt werden, in welchen Verantwortungsbereich die Abhilfe für die Fehlerursachen fällt. Weder kann ein Pilot die Fehlermeldung „Division durch 0“ auf seinem Bildschirm gebrauchen, weil jemand meinte, diesen Fehler durch eine solche Meldung endgültig behandeln zu können; noch hilft ihm die Meldung „Paritätsfehler“ weiter, die zustandekommen könnte, weil sich niemand für die Korrektur eines Datenübertragungsfehlers verantwortlich erklärte.

Wir verzichten in diesem Buch aus Platzgründen überwiegend auf Ausnahmebehandlungen. Wenn wir einen zum Abbruch führenden Fehler kennzeichnen wollen, schreiben wir die (in SATHER nicht vorgesehene) Fehleranweisung fehler.

Das Beispiel 8.12 zeigt, daß man anders als in Beispiel 8.11 eine Ausnahmebehandlung oft umgehen kann. Es ist schlechter Programmierstil, die Ausnahmebehandlung anstelle bedingter Anweisungen im normalen Programmieren einzusetzen. Andererseits sollte man einen einheitlichen Stil pflegen und Fehler nicht einmal wie in C mit Fehlernummer und ein anderes Mal mit Ausnahmebehandlung bearbeiten. Insbesondere gehört es zur Dokumentation von Prozeduren und Datenstrukturen, daß man sorgfältig aufschreibt, welche Fehler wie abgefangen werden, und welche möglichen Fehler unbearbeitet bleiben.

8.2 Zusicherungskalkül

Der vorige Abschnitt vermittelt ein informelles Verständnis zustandsorientierten Programmierens. Beispiele wie 8.4 oder 8.10 verlangen aber offensichtlich eine präzisere Verfolgung der durch Anweisungen bewirkten Zustandsänderungen, wenn wir uns nicht durch Tests, sondern „am Schreibtisch“ durch **symbolische Programmausführung** von der Richtigkeit des Programms überzeugen wollen.

Dies leistet der hier eingeführte Zusicherungskalkül. Er kann als formale Methode zum Korrektheitsbeweis von Algorithmen aufgefaßt und bis zu halbautomatischen Beweissystemen ausgebaut werden. In der praktischen Anwen-

dung liefert er ein Begriffsgerüst, um die Wirkung von Programmen während der Konstruktion zu erfassen. In dieser Form gebrauchen wir den Kalkül anschließend.

Es seien n Variable $v^{(1)}, \ldots, v^{(n)}$ vereinbart, die zusammen den Zustandsraum Z unserer Berechnung aufspannen. Die Werte $w^{(1)}, \ldots, w^{(n)}$ dieser Variablen bilden den jeweiligen Zustand z. Eingabedaten seien durch die Anfangswerte der Variablen im Zustand z_0 gegeben. Wir betrachten Anweisungen und Anweisungsfolgen A, die die Werte dieser Variablen verändern. Den Wert einer Variablen v im Anfangszustand z_0 bezeichnen wir mit v_0. Den Wert zu Beginn einer Anweisung A bezeichnen wir häufig mit $v_v = v_{vorher}$, den Wert nach Ausführung von A mit $v_n = v_{nachher}$. Der Zustand z ist durch das Prädikat

$$P\colon (v^{(1)} = w^{(1)}) \wedge \cdots \wedge (v^{(n)} = w^{(n)}) \tag{8.14}$$

genau beschrieben.

Bei symbolischer Programmausführung wollen wir uns den Ablauf eines Programms für *beliebige* Eingabedaten klarmachen. Dabei können wir die einzelnen Zustände nicht so genau charakterisieren wie in (8.14). Jedoch könnte es abgeschwächte Bedingungen P' geben, denen die Variablenwerte in einem bestimmten Zustand unabhängig von den Eingabedaten genügen. Wenn die Eingabedaten z. B. von einer Meßapparatur stammen, die nur Werte x im Zahlbereich $a \leqslant x \leqslant b$ liefern kann, kennen wir zwar den exakten Wert von x nicht. Wohl aber können wir zusichern, daß der Wert im Intervall $[a, b]$ liegt.

Ein Prädikat P, das Eigenschaften eines Zustands z entweder präzise wie in (8.14) oder in abgeschwächter Form beschreibt, heißt eine **Zusicherung**[23] über den Zustand z. P beschreibt die Menge

$$Z(P) = \{z \mid P(z)\} \tag{8.15}$$

aller Zustände z, für die P gilt. Wir sprechen kurz vom „Zustand P“, wenn wir einen beliebigen, aber festen Zustand z bezeichnen wollen, für den die Zusicherung $P(z)$ gilt.

Zusicherungen sind prädikatenlogische Formeln nach den Regeln aus Abschnitt 4.2. Aus $P \models Q$ oder $P \rightarrow Q$ folgt also, daß ein Zustand z, für den die Zusicherung P gilt, auch die Zusicherung Q erfüllt. $P(z)$ bedeutet, daß es eine Kombination z von Variablenwerten gibt, die P erfüllt. Der Zustand z liefert ein Modell für P; daraus folgt nicht, daß ein Programm beginnend mit geeigneten Anfangswerten den Zustand z erreichen kann. Gilt umgekehrt $Z(P) = \emptyset$, so ist P während der Ausführung eines Programms bestimmt nicht erfüllbar.

Sei A ein Programm mit Variablen $v^{(1)}, \ldots, v^{(n)}$, dessen Ausführung eine Folge von Zuständen $z_0, z_1, \ldots, z_n, \ldots$ durchläuft. Wir unterscheiden zwei Aufgabenstellungen, die aufeinander aufbauen:

23. engl. *assertion*.

- Zustandsverfolgung: Wir wollen Zusicherungen $Q_n = Q(z_n)$ über einen Zustand z_n ermitteln, wenn eine Zusicherung $P_0 = P(z_0)$ über den Anfangszustand z_0 vorliegt. Oft stellen wir die Frage auch umgekehrt: Gegeben sei $Q(z)$. Welche Zusicherung $P_0 = P(z_0)$ gewährleistet die Existenz eines n, so daß nach n Schritten $Q_n = Q(z)$ für den dann erreichten Zustand gilt?
- Verifikation: Gegeben sei ein Paar (P, Q) von Zusicherungen über mögliche Zustände unseres Programms A. Wir nennen (P, Q) eine **Spezifikation**. A heißt **partiell korrekt**, wenn es ein n gibt, so daß $P(z_0)$ und $Q(z_n)$ gelten unter der Annahme, daß das Programm terminiert. Es heißt spezifikationstreu oder **total korrekt**, wenn es partiell korrekt ist und nach n Schritten tatsächlich terminiert. Diese Definitionen decken sich mit den entsprechenden Definitionen in Abschnitt 5.5.1. Die Nachprüfung der partiellen oder totalen Korrektheit eines Programms heißt **Verifikation** des Programms (bezüglich der Spezifikation (P, Q)).

Alle Axiome abstrakter Datentypen aus Kap. 6 sind Spezifikationen in diesem Sinne: Das Kelleraxiom

```
pop (Push k x) = k
```

geht von einem Anfangszustand aus, für den die Zusicherung P: *k ist ein Keller mit Wert* k_0 *und x gehört zum Elementtyp T des Kellers* gilt. Die Zusicherung Q lautet: *nach Ausführung von* `k' := Push k x` *und* `pop k'` *gilt* $k = k_0$. Hier ist sogar n mit angegeben: Nach 2 Operationen erreichen wir wieder den Anfangszustand.

Ist eine Abbildung $f: X \to Y$ zwischen zwei Wertebereichen X, Y gegeben, so lautet die Spezifikation eines Programms A mit Variablen x, y zur Berechnung von $y = f(x)$: $(P: x = x_0 \in X,\ Q: y = f(x_0))$. Funktionsspezifikationen sind ein Spezialfall der Spezifikation einer beliebigen Relation $\rho \subseteq X \times Y$; die Spezifikation lautet dann $(P: x_0 \in X, Q: x_0\ \rho\ y)$. Es kann auch spezifiziert werden, daß f auf X nur partiell definiert ist und für die Werte $x_0 \in X$, für die f nicht definiert ist, das Programm Fehler meldet, d. h. die Fehleranweisung ausführt, oder daß es in einem solchen Fall nicht terminiert.

Die Verifikation bezieht sich auf die **funktionale Korrektheit** eines Programms. Damit es praktisch brauchbar ist, muß ein Programm meist weitere **nicht-funktionale Eigenschaften** besitzen: ein Steuerprogramm muß Zeitbedingungen einhalten, ein Texteditor muß eine „angemessene" Benutzeroberfläche bereitstellen, usw. Unter der **Validierung** eines Programms versteht man die Überprüfung aller vom Benutzer gewünschten Eigenschaften einschließlich der nicht-funktionalen. Validierung ist also allgemeiner als Verifikation. Forderungen wie „angemessene Benutzeroberfläche" lassen sich nicht mathematisch spezifizieren; Validierung kann daher kein Beweis im Sinne der Mathematik sein, sondern kann höchstens solche Beweise umfassen.

Beispiel 8.13: Für das Beispiel 8.5, S. 22, der Berechnung des ggT betrachten wir das Prädikat

$$P : a > 0 \wedge b \geqslant 0 \wedge \mathrm{ggT}(a, b) = \mathrm{ggT}(a_0, b_0) \wedge (a, b \text{ ganz})$$

mit den Anfangswerten a_0, b_0 für die Variablen a, b. P ist zu Anfang richtig. Wegen $(b \neq 0) \rightarrow (\mathrm{ggT}(a, b) = \mathrm{ggT}(b, a \bmod b))$ gilt P auch nach einer Ausführung des Schleifenrumpfes; man sieht, daß P eine Schleifeninvariante ist; der Schleifenrumpf

h: INT $:= a; a := b; b := h \bmod b$

stellt die Bedingung P wieder her. Wegen $(b = 0) \rightarrow (\mathrm{ggT}(a, b) = a)$ ergibt sich das gewünschte Ergebnis $a = \mathrm{ggT}(a_0, b_0)$, wenn die Schleife anhält. Für $a_0 > 0$ ist die Folge $a_1, a_2, \ldots$ streng monoton fallend und wegen der Gültigkeit von P durch 0 nach unten beschränkt; daher gibt es ein n so, daß das Programm nach n Schritten anhält. Das Programm ist also total korrekt. ♦

In der Praxis ist es nahezu unmöglich, ein bereits geschriebenes Programm nachträglich zu verifizieren. Wir müssen die Menge $\mathcal{P}$ von Zusicherungen mit den angegebenen Eigenschaften bereits während der Konstruktion angeben. Verifikation ist in Wahrheit ein Hilfsmittel der Programmkonstruktion. Dazu benutzen wir den von Hoare (1969) eingeführten und auf Gedanken von R. Floyd aufbauenden **Zusicherungskalkül**:

Ist A eine Anweisung einer imperativen Programmiersprache, z. B. eine Zuweisung, und beschreiben die Zusicherungen P bzw. Q den Zustand der Programmausführung vor und nach Ausführung der Anweisung A, so sagen wir, daß die Zusicherung

$$\{P\}\ A\ \{Q\} \tag{8.16}$$

gilt, wenn die Ausführung von A in einem durch P beschriebenen Zustand nach endlich vielen Schritten zu dem durch Q beschriebenen Zustand führt. Eine Zuweisung $i := 7$ liefert z. B.

$$\{P\text{: } i \text{ beliebig}\}\ i := 7\ \{Q\text{: } i = 7\}. \tag{8.17}$$

P heißt **Vor**- und Q **Nachbedingung** einer solchen Anweisung.[24]

P und Q sind Zusicherungen über Zustände; $\{P\}\ A\ \{Q\}$ ist eine Zusicherung über den Ablauf. Der Zusatz „nach endlich vielen Schritten“ verlangt, daß Schleifen und rekursive Prozeduraufrufe terminieren.

Die Anweisung A kann auch das ganze Programm sein. Das Paar P, Q spezifiziert dann die Aufgabe; A löst sie, wenn (8.16) gilt. (8.16) ist eine andere Formulierung der totalen Korrektheit.

24. engl. *precondition* und *postcondition*. Wir schreiben „Vor:“ bzw. „Nach:“ (im Englischen “*pre*:” bzw. “*post*:”), um Vor- und Nachbedingungen zu kennzeichnen.

(8.16) ist ebenso wie P und Q eine logische Formel, die wahr oder falsch sein kann. Insbesondere gilt:

$$\text{Aus } P' \to P,\ Q \to Q' \text{ und } \{P\}\ A\ \{Q\} \text{ folgt } \{P'\}\ A\ \{Q'\}. \tag{8.18}$$

Vorbedingungen können verschärft, Nachbedingungen abgeschwächt werden.

Die Zusicherung $\{P\}\ A\ \{Q\}$ gewährleistet, daß die Anweisung A terminiert. HOARE benutzte ursprünglich die Schreibweise

$$P\ \{A\}\ Q \tag{8.19}$$

und bezeichnete damit die partielle Korrektheit von A von S. 35: Falls A im Zustand P ausgeführt wird und terminiert, so gilt anschließend Q. Mit der Notation $P\ \{A\}\ Q$ müssen wir das Terminieren von A gesondert beweisen.

Nach der Ausführung einer Zuweisung $j := 2$ gilt $\{j = 2\}$. Diese Zusicherung wird durch die anschließende Zuweisung (8.17) nicht verändert. Die Nachbedingung zu $j := 2;\ i := 7$ lautet $Q: (j = 2) \wedge (i = 7)$. Hätten wir als erstes jedoch nicht $j := 2$, sondern $i := 2$ ausgeführt, so wäre die Nachbedingung in (8.17) unverändert und ohne Erweiterung übernommen worden, da bei der Zuweisung $i := 7$ die Wirkung einer vorangehenden Zuweisung an i verlorengeht. Bei der Feststellung der Nachbedingung interessiert uns also nicht nur, was die einzelne Anweisung bewirkt, sondern zusätzlich, welche Teile der Vorbedingung übernommen bzw. verändert werden.

Nachbedingungen können auch uninteressante Aussagen umfassen: Nach der Zuweisungsfolge $j := 2;\ i := j + 1$ bleibt offen, ob wir in der Nachbedingung $Q: (j = 2) \wedge (i = 3)$ die Aussage über j wirklich noch benötigen; die Zuweisung an j könnte ausschließlich das Ziel gehabt haben, die spätere Berechnung von i zu ermöglichen. In diesem Fall wäre die Zusicherung $j = 2$ im weiteren Verlauf uninteressant und könnte weggelassen werden (Abschwächung der Nachbedingung). Wenn wir das Programm von vorne nach hinten durchgehen, wissen wir aber noch nicht, ob wir j noch benötigen.

Der Kalkül der **Vorwärtsanalyse**, bei dem wir wie eben skizziert vorgehen, hat also den Nachteil, daß er nicht zielorientiert ist; die Zusicherung, die wir am Programmende erreichen, kann viele Aussagen umfassen, die zum Korrektheitsnachweis überflüssig sind.

Eine **Rückwärtsanalyse** vermeidet diesen Nachteil der Vorwärtsanalyse und ist damit zielorientiert: Sie geht von der Nachbedingung Q des Programms aus, die das gewünschte Gesamtergebnis charakterisiert, und fragt, unter welcher Vorbedingung P die Anweisung oder das Programm A die gewünschte Nachbedingung Q liefert. Natürlich könnte es viele Vorbedingungen geben, unter denen das gewünschte Ergebnis erreicht werden kann. Die Abb. 8.5 zeigt die Situation.

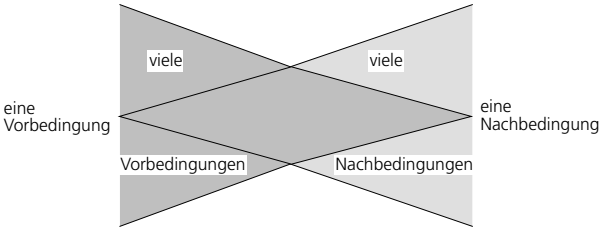

Abbildung 8.5: Vor- und Rückwärtsanalyse

Vor- und Rückwärtsanalyse sind auch in zahlreichen anderen Wissenschaften bekannt: In der numerischen Mathematik können wir entweder aus der Genauigkeit der Eingabedaten auf die Genauigkeit der Ergebnisse schließen. Oder wir können aus der gewünschten Genauigkeit des Ergebnisses rückwärts schließen, wie genau die Eingabedaten sein müssen. Analoge Überlegungen kann man bei der Auswertung physikalischer Experimente anstellen. In der Geschichts-, Politik- und Wirtschaftswissenschaft kann man entweder aus beobachteten gesellschaftlichen Verhältnissen auf die zukünftige Entwicklung schließen wollen, oder man kann aus beobachteten oder gewünschten Verhältnissen auf die Einflußfaktoren zu schließen versuchen, die diese Verhältnisse herbeigeführt haben oder herbeiführen könnten.

Die Menge aller möglichen Vorbedingungen ist halbgeordnet: die Vorbedingung P ist schwächer als P', wenn $P' \to P$ gilt. Insbesondere ist die Disjunktion $P' \vee P''$ zweier Vorbedingungen P', P'' schwächer als jede von ihnen, und allgemein ist die Disjunktion *aller* möglichen Vorbedingungen schwächer als jede einzelne solche Bedingung. Damit haben wir eine Möglichkeit, statt vieler Vorbedingungen eine einzelne zu untersuchen, nämlich die schwächste. Bei vorgegebener Nachbedingung Q und Anweisung A bezeichnet man sie mit

$$P = \mathrm{wp}(A, Q). \tag{8.20}$$

Der Übergang zur schwächsten Vorbedingung[25] heißt **Prädikattransformation**. $\mathrm{wp}(A, Q)$ heißt die Prädikattransformierte von Q durch die Anweisung A.

Damit haben wir zwei eng verwandte **Zusicherungskalküle** in der Hand, um die Korrektheit eines Programms nachzuweisen:

- Wir können ein Programm am Anfang und nach jeder Anweisung, insbesondere auch am Ende, mit Zusicherungen versehen und dann durch Betrachtung jeder einzelnen Anweisung A nachweisen, daß A tatsächlich seine Vorbedingung in seine Nachbedingung überführt. Diese Vorgehensweise ist als Anwendung der **Floyd-Hoare-Logik** oder kürzer der **Hoare-Logik** bekannt.
- Wir können aber auch mit der Endebedingung des Gesamtprogramms beginnen und mit dem auf E. W. Dijkstra zurückgehenden **Kalkül der schwächsten Vorbedingungen** oder kurz **wp-Kalkül** die Anweisungen des Programms

25. engl. *weakest precondition*.

rückwärts untersuchen, um jeweils die schwächste Vorbedingung festzustellen. Diese ist dann zugleich Nachbedingung der vorangehenden Anweisung. Das Programm ist total korrekt, wenn die ursprüngliche Spezifikation der Eingabedaten die schwächste Vorbedingung des Gesamtprogramms erfüllt.

In beiden Fällen benötigen wir für jede einzelne Anweisung A einen Beweis, daß (8.16) bzw. (8.20) gilt. Dazu geben wir in den nachfolgenden Abschnitten Axiome für die Anweisungsarten aus dem vorigen Abschnitt an. Beide Kalküle benutzen im wesentlichen die gleichen Axiome; nur für Schleifen gibt es unterschiedliche Gesetze. Hier zeigt sich, daß der Kalkül der schwächsten Vorbedingungen allgemeiner ist. In der Praxis genügt uns jedoch gewöhnlich HOARE-Logik: Wir benutzen unsere Vorkenntnisse über den geplanten Programmablauf, um die Nachbedingungen auf die notwendigen Aussagen zu beschränken.

Für schwächste Vorbedingungen gibt es einige elementare Gesetze:

Satz 8.1 (**Wunder sind ausgeschlossen**):

$$\mathrm{wp}(A, \mathit{falsch}) = \mathit{falsch}.$$

Da *falsch* nicht erfüllbar ist, muß auch die zugehörige Vorbedingung unerfüllbar sein. Andernfalls hätten wir eine Anweisung, die aus einem erfüllbaren Zustand in einen unerfüllbaren Zustand, also einen „Nicht-Zustand“ führt.

Satz 8.2:

Distributivität der Konjunktion:

$$\mathrm{wp}(A, Q) \wedge \mathrm{wp}(A, R) \quad = \quad \mathrm{wp}(A, Q \wedge R), \tag{8.21}$$

Distributivität der Disjunktion:

$$\mathrm{wp}(A, Q) \vee \mathrm{wp}(A, R) \quad \rightarrow \quad \mathrm{wp}(A, Q \vee R), \tag{8.22}$$

Monotoniegesetz:

$$\textit{Aus } Q \rightarrow R \quad \textit{folgt} \quad \mathrm{wp}(A, Q) \rightarrow \mathrm{wp}(A, R). \tag{8.23}$$

Zum Beweis von (8.21) sei z ein Zustand, der die schwächsten Vorbedingungen von Q und R, also die linke Seite von (8.21) erfüllt. Dann ist $Q \wedge R$ nach Ausführung von A wahr. Es gilt $\mathrm{wp}(A, Q) \wedge \mathrm{wp}(A, R) \rightarrow \mathrm{wp}(A, Q \wedge R)$. Erfüllt umgekehrt z die Bedingung $\mathrm{wp}(A, Q \wedge R)$, dann erfüllt es wegen (8.18) auch $\mathrm{wp}(A, Q)$ und $\mathrm{wp}(A, R)$. Dies beweist die umgekehrte Implikation.

(8.22) beweist man ebenso. Die Umkehrung von (8.22) gilt nicht: Werfen wir z. B. eine Münze, so bleibt diese bestimmt mit Zahl oder Wappen nach oben liegen. Es gilt also

$$\mathrm{wp}(\text{„wirf Münze“}, \text{Zahl} \vee \text{Wappen}).$$

Bei diesem zufallsabhängigen Experiment gibt es jedoch keine Vorbedingung dafür, daß auf jeden Fall die Zahl oben liegt. Es gilt also wp(„wirf Münze“, Zahl) = *falsch* und ebenso wp(„wirf Münze“,Wappen) = *falsch*, insgesamt also wp(„wirf Münze“, Zahl) ∨ wp(„wirf Münze“, Wappen) = *falsch*.

Die Aussage (8.23) folgt aus (8.18). ♦

Daß wir für (8.22) nur die eine Richtung beweisen können, ist ausschließlich eine Folge von Indeterminismus. Bei deterministischem Programmablauf läßt sich für jeden Zustand z genau vorhersagen, welcher Zustand z_n durch Ausführung der Anweisung A erreicht wird. Erfüllt z die Bedingung $\mathrm{wp}(A, Q \vee R)$, dann muß z_n mindestens eine der Bedingungen Q oder R erfüllen. Im ersten Fall erfüllt z auch $\mathrm{wp}(A, Q)$, im zweiten Fall erfüllt es $\mathrm{wp}(A, R)$: eine der beiden Bedingungen ist also immer erfüllt und es gilt $\mathrm{wp}(A, Q \vee R) \rightarrow \mathrm{wp}(A, Q) \vee \mathrm{wp}(A, R)$. Wir haben also:

Korollar 8.3: *In deterministischen Programmiersprachen gilt die Distributivität der Disjunktion:*

$$\mathrm{wp}(A, Q) \vee \mathrm{wp}(A, R) = \mathrm{wp}(A, Q \vee R). \tag{8.24}$$

In Sather und ähnlich in Eiffel sowie Java ab Version 1.5 kann man Zusicherungen in der Form

assert *boolescher Ausdruck* (8.25)

zwischen zwei Anweisungen explizit in das Programm schreiben. Bei Programmausführung wird der boolesche Ausdruck berechnet. Die Ausführung wird nur fortgesetzt, wenn das Ergebnis *wahr* ist; andernfalls wird die Ausnahme ASSERTION_ERROR ausgelöst.

In einfachen Fällen, z. B. für die meisten Beispiele dieses Kapitels, lassen sich damit Zusicherungen automatisch überprüfen. Für anspruchsvollere Probleme ist das Verfahren jedoch nicht ausreichend: Einerseits erreichen die booleschen Ausdrücke in Programmiersprachen nicht die volle Allgemeinheit des Prädikatenkalküls; insbesondere wird für eine Formel $\forall x : P(x)$ immer nur die Gültigkeit von P für den aktuellen Wert von x geprüft. Andererseits verlangt die Überprüfung einer Zusicherung $\{P\}\ A\ \{Q\}$ gewöhnlich, daß zur Berechnung von Q der Wert x_v bestimmter Variabler x vor Ausführung von A noch zur Verfügung steht. Diesen kann man sich zwar merken, wenn es sich um eine ganzzahlige Variable handelt. Ist aber $\{P\}\ A\ \{Q\}$ eine Zusicherung über eine Veränderung einer Datenbank x, dann kann man deren alten Wert x_v für eine Berechnung von Q gewöhnlich nicht mehr mit akzeptablem Aufwand zur Verfügung stellen.

Wir schreiben nachfolgend oft Zusicherungen in der Form $\{P\}$ oder als Kommentare in unsere Programme.

8.2.1 Axiome des Zusicherungskalküls

Wir betrachten nacheinander die Zuweisung und die Anweisungsarten aus Abschnitt 8.1.6 und geben Axiome an, denen die Übergänge zwischen Vor- und Nachbedingung bei Ausführung solcher Anweisungen in der Hoare-Logik bzw. dem Kalkül der schwächsten Vorbedingungen genügen. Da wir bisher die Bedeutung der einzelnen Anweisungen nur informell erklärt haben, liegt uns keine

Spezifikation vor, wie wir sie für einen Korrektheitsbeweis benötigen. Wir können also nur inhaltlich argumentieren, daß unsere Axiome „vernünftig“ sind. In Wahrheit definieren wir mit den Axiomen die Semantik der Anweisungen.

Dies zeigt sich auch daran, daß diese Axiome nicht nur zum Nachweis der Korrektheit eines Programms, sondern auch zum Nachweis der Korrektheit der Implementierung der Programmiersprache herangezogen werden können. Diese muß, im Sinne der Logik, ein Modell der durch die Axiome definierten Theorie sein.

Leider betrachten viele Programmierer die Implementierung der jeweiligen Programmiersprache als Definition der Semantik. Sie verschwenden Zeit damit, herauszufinden, welchen Programmiertricks ein Übersetzer welche Bedeutung zuordnet. Der Gebrauchswert solcher Tricks ist allerdings beschränkt, da sie beim Übergang zu einer anderen Implementierung meist ungültig werden. Nur, wenn die Sprachelemente entsprechend ihrer hier axiomatisch angegebenen Definition eingesetzt werden, kann man ein Programm wirklich als *sinnvoll* bezeichnen.

Wir geben die Axiome jeweils in den beiden Schreibweisen der HOARE-Logik und des wp-Kalküls an.

Die Leeranweisung leer bewirkt keine Zustandsänderung. Daher gelten vorher und nachher die gleichen Zusicherungen:

Axiom der Leeranweisung:

$$\{P\} \quad \mathsf{leer} \quad \{P\}, \tag{8.26}$$

$$\mathrm{wp}(\mathsf{leer}, P) \quad = \quad P. \tag{8.27}$$

Die Fehleranweisung fehler ist der Versuch, eine Anweisung auszuführen, deren Vorbedingung nicht erfüllt ist. Die Vorbedingung ist also *falsch*, unabhängig von der Nachbedingung:

Axiom der Fehleranweisung:

$$\{\mathit{falsch}\} \quad \mathsf{fehler} \quad \{P\}, \tag{8.28}$$

$$\mathrm{wp}(\mathsf{fehler}, P) \quad = \quad \mathit{falsch}. \tag{8.29}$$

8.2.2 Zuweisung

Eine Zuweisung $v := a$ des Ergebnisses eines Ausdrucks a an eine Variable v verändert nur Aussagen über v; alle anderen Aussagen bleiben unverändert. Wie wir bereits auf S. 4 feststellten, gelten über v in der Nachbedingung alle Aussagen, die zuvor über den (Wert des) Ausdruck a galten. Dieser muß berechenbar sein, es muß also $\mathsf{zulässig}(a)$ gelten. Genauso muß der Name v, der z. B. Indizes enthalten kann, berechenbar sein. Ist Q die Nachbedingung, so ist die schwächste Vorbedingung $\mathrm{wp}(\text{„}v := a\text{“}, Q)$ folglich $\mathsf{zulässig}(v) \wedge \mathsf{zulässig}(a) \wedge Q[a/v]$: Wir substituieren in Q überall den Ausdruck a für die Variable v, um die Vorbedingung zu erhalten. In Formelschreibweise lautet dieses Axiom

Zuweisungsaxiom:

$$\{(\text{zulässig}(v) \wedge \text{zulässig}(a)) \wedge Q[a/v]\} \quad v := a \quad \{Q\}, \tag{8.30}$$

$$\text{wp}(\text{„}v := a\text{“}, Q) \;=\; (\text{zulässig}(v) \wedge \text{zulässig}(a)) \wedge Q[a/v]. \tag{8.31}$$

In der Literatur über Programmkorrektheit findet man häufig die Notationen Q_a^v oder $Q_{a \to v}$ statt $Q[a/v]$.

Die Zulässigkeit von a und v ist notwendig, um $Q[a/v]$ zu bestimmen. Daher muß die Konjunktion mit Kurzauswertung berechnet werden. Sehr häufig ist die Zulässigkeit der Ausdrücke unmittelbar einsichtig. Wir benutzen dann das Zuweisungsaxiom in der verkürzten Fassung

$$\{Q[a/v]\}\ v := a\ \{Q\} \tag{8.32}$$

bzw.

$$\text{wp}(\text{„}v := a\text{“}, Q) = Q[a/v]. \tag{8.33}$$

Beispiel 8.14: Ist Q: $v = m$ die gewünschte Nachbedingung für die Zuweisung $v := v + 1$, so ergibt das Zuweisungsaxiom:

$$\begin{aligned} \text{wp}(\text{„}v := v + 1\text{“}, v = m) &= Q[(v + 1)/v] \\ &= (v + 1 = m). \end{aligned}$$

Auflösung nach v ergibt die Vorbedingung $v = m - 1$. ♦

Beispiel 8.15: Für ein beliebiges einstelliges Prädikat p gilt

$$\text{wp}(\text{„}v := a \text{ div } b\text{“}, \{p(v)\}) = \{b \neq 0 \wedge p(a \text{ div } b)\}$$

Der Zusatz $b \neq 0$ resultiert aus der Forderung zulässig(a). ♦

Aufgabe 8.10: Berechnen Sie:

1. $\text{wp}(\text{„}x := x * y\text{“}, x * y = c)$
2. $\text{wp}(\text{„}x := (x - y) * (x + y)\text{“}, x + y^2 \neq 0)$

Beispiel 8.16 (**Zuweisung an Reihungselemente**): Den Wert einer Reihung a nach einer Zuweisung $a[i] := w$ beschreiben wir wie auf S. 13 durch $(a\colon [w/i])$. Abgesehen von der Prüfung der Zulässigkeit des Index von i, also $0 \leqslant i < a.\text{asize}$, und der Berechnung von w, folgt daher aus (8.31) $\text{wp}(\text{„}a[i] := w\text{“}, Q) = Q[(a\colon [w/i])/a]$. Damit ergibt sich für die Zuweisung $a[i] := 5$:

$$\begin{aligned} &\text{wp}(\text{„}a[i] := 5\text{“}, a[i] = a[j]) \\ &= (a[i] = a[j])[(a\colon [5/i])/a] && \text{Zuweisung Reihungselement} \\ &= (a[i])[(a\colon [5/i])/a] = (a[j])[(a\colon [5/i])/a] && \text{Substitution} \\ &= (i \neq j \wedge 5 = a[j]) \vee (i = j \wedge 5 = 5) && \text{Fallunterscheidung: } i = j \vee i \neq j \\ &= (i \neq j \wedge 5 = a[j]) \vee (i = j) \\ &= (i \neq j \vee i = j) \wedge (5 = a[j] \vee i = j) && \text{Distributivgesetz} \\ &= \mathit{wahr} \wedge (i = j \vee a[j] = 5) \\ &= (i = j \vee a[j] = 5). \end{aligned}$$

Wenn wir wie auf S. 13 eine Reihung als Funktion ansehen, ergibt sich die Fallunterscheidung zwangsläufig. ♦

Beispiel 8.17: Die Zuweisung „$a[a[i]] := i$“ hat keine Auswirkung auf die Zusicherung $a[i] = i$:

$$\begin{array}{lll}
\text{wp}(\text{„}a[a[i]] := i\text{“}, a[i] = i) & & \\
\quad = & (a[i] = i)[(a\colon [i/a[i]))/a] & \text{Zuweisung} \\
\quad = & (a[i])[(a\colon [i/a[i]))/a] = i & \text{Substitution} \\
\quad = & (a[i] \neq i \wedge a[i] = i) \vee (a[i] = i \wedge i = i) & \text{Fallunterscheidung} \\
\quad = & \mathit{falsch} \vee (a[i] = i \wedge \mathit{wahr}) & \\
\quad = & a[i] = i. & \blacklozenge
\end{array}$$

Aufgabe 8.11: Berechnen Sie: wp(„$a[i] := i$“, $a[a[i]] = i$)

8.2.3 Hintereinanderausführung, Blöcke

Mit der Hintereinanderausführung $A_1; A_2$ zweier Anweisungen erreichen wir eine Nachbedingung R aus einer gegebenen Vorbedingung P, wenn es eine Zusicherung Q gibt mit $\{P\}\ A_1\ \{Q\}$ und $\{Q\}\ A_2\ \{R\}$. Dies gilt auch für die schwächsten Vorbedingungen und wir erhalten:

Axiom des sequentiellen Ablaufs:

$$\{P\}\ A_1; A_2\ \{R\} \quad \asymp \quad \exists Q : \{P\}\ A_1\ \{Q\},\ \{Q\}\ A_2\ \{R\}, \tag{8.34}$$

$$\text{wp}(\text{„}A_1; A_2\text{“}, R) \quad = \quad \text{wp}(\text{„}A_1\text{“}, \text{wp}(\text{„}A_2\text{“}, R)). \tag{8.35}$$

Aufgabe 8.12: Zeigen Sie, daß aus $\{P\}\ A_1\ \{Q_1\}$, $\{Q_2\}\ A_2\ \{R\}$ und $Q_1 \to Q_2$ folgt $\{P\}\ A_1; A_2\ \{R\}$.

Aufgabe 8.13 (**Gries**): Zeigen Sie, daß (8.35) der Bedingung (8.24) genügt, wenn A_1 und A_2 die Eigenschaft (8.24) haben. Sequentielle Verknüpfung kann keinen Indeterminismus verursachen.

Aufgabe 8.14 (**Gries**): Zeigen Sie wp(„A; fehler“, R) = *falsch* für beliebige Anweisungen A.

Beispiel 8.18: In Abschnitt 8.1.6.1 haben wir gezeigt, daß die Werte i_0, j_0 zweier Variabler i, j durch die Anweisungsfolge $h := i;\ i := j;\ j := h$ mit einer Hilfsvariablen h vertauscht werden können. Hierfür lautet die Spezifikation

$$\begin{array}{rl}
\text{Vor:} & i = i_0 \wedge j = j_0 \\
\text{Nach:} & i = j_0 \wedge j = i_0
\end{array}$$

Wenn wir die Vor- und Nachbedingung für das Programmstück eintragen, erhalten wir

$$\begin{array}{c} \{i = i_0 \wedge j = j_0\} \\ h := i; \\ i := j \\ j := h \\ \{i = j_0 \wedge j = i_0\} \end{array} \tag{8.36}$$

Ein solches Schema nennt man eine **Beweisvorgabe**[26]. Eine Beweisvorgabe kann auch alle elementaren Zwischenschritte enthalten.

$$\begin{array}{c} \{i = i_0 \wedge j = j_0\} \\ h := i; \\ \{i = i_0 \wedge j = j_0 \wedge h = i_0\} \\ i := j \\ \{i = j_0 \wedge j = j_0 \wedge h = i_0\} \\ j := h \\ \{i = j_0 \wedge j = i_0 \wedge h = i_0\} \end{array} \tag{8.37}$$

Hier haben wir zu den Anweisungen die Zusicherungen geschrieben, die wir bei Vorwärtsanalyse nach einer Anweisung erreichen.

Aufgabe 8.15: Überprüfen Sie mit Hilfe von (8.32) die einzelnen Übergänge.

Im wp-Kalkül lautet dieses Beispiel

$$\begin{aligned} & \mathrm{wp}(\text{„}h := i; i := j; j := h\text{“}, i = j_0 \wedge j = i_0) \\ = \quad & \mathrm{wp}(\text{„}h := i; i := j\text{“}, \mathrm{wp}(\text{„}j := h\text{“}, i = j_0 \wedge j = i_0)) \\ = \quad & \mathrm{wp}(\text{„}h := i; i := j\text{“}, i = j_0 \wedge h = i_0) \\ = \quad & \mathrm{wp}(\text{„}h := i\text{“}, \mathrm{wp}(\text{„}i := j\text{“}, i = j_0 \wedge h = i_0)) \\ = \quad & \mathrm{wp}(\text{„}h := i\text{“}, j = j_0 \wedge h = i_0) \\ = \quad & \{j = j_0 \wedge i = i_0\} \end{aligned}$$

Dabei haben wir zweimal hintereinander (8.35) und (8.33) angewandt. Den Term $h = i_0$ in der abschließenden Nachbedingung von (8.37) benötigen wir nicht. Die Aussage ist zwar richtig, aber unwichtig. Bei Vorwärtsanalyse ergibt sie sich dennoch automatisch. Dies illustriert nochmals, daß die Vorwärtsanalyse unter Umständen uninteressante Aussagen mitschleppt, wenn wir beim Beweis die beabsichtigte Nachbedingung nicht berücksichtigen. ♦

Beispiel 8.19: In Beispiel 8.18 haben wir nirgends vom Typ der Variablen i, j, h Gebrauch gemacht. Der Beweis ist also für Variable beliebigen Typs richtig. Für

26. engl. *proof obligation*.

ganzzahlige Variable können wir die Aufgabe aus dem vorigen Beispiel auch ohne Hilfsvariable lösen. Es gilt nämlich:

$$\begin{aligned}
&\mathrm{wp}(\text{„}i := i - j; j := i + j; i := j - i;\text{“}, j = i_0 \wedge i = j_0)\\
&\quad = \quad \mathrm{wp}(\text{„}i := i - j; j := i + j;\text{“}, \mathrm{wp}(\text{„}i := j - i;\text{“} j = i_0 \wedge i = j_0))\\
&\quad = \quad \mathrm{wp}(\text{„}i := i - j; j := i + j;\text{“}, j = i_0 \wedge i = i_0 - j_0)\\
&\quad = \quad \mathrm{wp}(\text{„}i := i - j;\text{“}, \mathrm{wp}(\text{„}j := i + j;\text{“}, j = i_0 \wedge j - i = j_0))\\
&\quad = \quad \mathrm{wp}(\text{„}i := i - j;\text{“}, i + j = i_0 \wedge i + j - i = j_0)\\
&\quad = \quad \{i - j + j = i_0 \wedge j = j_0\}\\
&\quad = \quad \{i = i_0 \wedge j = j_0\}
\end{aligned}$$

♦

Aufgabe 8.16: Formulieren Sie die Beweisvorgabe für Beispiel 8.19 und beweisen Sie sie mit Vorwärtsanalyse. Geben Sie ein Beispiel an, in dem 8.19 bei Zugrundelegung üblicher Rechnerarithmetik fehlschlägt. Welche Nachlässigkeit haben wir begangen, die uns diesen Fehler übersehen ließ, und wie können wir das korrigieren?

Aufgabe 8.17: Warum sollte Beispiel 8.19 nicht auf Gleitpunktzahlen angewandt werden?

Aufgabe 8.18: Können Sie die Werte von booleschen Variablen ohne Hilfsvariable vertauschen? Beweisen Sie die Korrektheit ihrer Lösung.

Aufgabe 8.19: Zeigen Sie die Zusicherung

$\{a > 0 \wedge b > 0 \wedge \mathrm{ggT}(a, b) = x\}$
$h := a;\ a := b;\ b := h \bmod b$
$\{a > 0 \wedge \mathrm{ggT}(a, b) = x\}$

Die Anweisungsfolge läßt Aussagen über den ggT unverändert, nur die Aussage $b > 0$ könnte sich ändern. Zusammen mit Beispiel 8.13 beweist dies die Korrektheit von Beispiel 8.5.

Wenn wir eine Anweisungsfolge A zu einem Block begin h: T; A end erweitern, indem wir eine oder mehrere Vereinbarungen h: T voranstellen, ändert sich an unseren Axiomen wenig: Unter Berücksichtigung der Gültigkeitsbereichsregeln aus Abschnitt 8.1.3 wissen wir, daß in Vor- und Nachbedingungen des Blocks die Größe h nicht vorkommen kann. Bei Vorwärtsanalyse lassen wir daher am Blockende etwaige Zusicherungen über die lokale Größe h einfach weg. Bei Rückwärtsanalyse ergibt sich wp(„begin h: T; A end“, Q) = *falsch*, wenn h in wp(„A“, Q) vorkommt: Über den Wert von h zu Beginn des Blocks kann keine Aussage gemacht werden. Eine vorbesetzende Vereinbarung h: $T := a$ wird wie die Folge h: T; $h := a$ behandelt.

8.2.4 Bedingte Anweisungen

Wie in den Abschnitten 8.1.6.2 und 8.1.6.3 setzen wir voraus, daß die Berechnung der Bedingung b einer bedingten Anweisung oder einer while-Schleife keine Nebenwirkungen hat. Die Forderung *nebenwirkungsfrei* schließt die Forderung zulässig(b) ein.

Wir wissen bereits, daß die einseitige bedingte Anweisung if b then A end äquivalent ist zu

if b then A else leer end

In funktionalen Sprachen bedeutet sie allerdings if b then A else fehler end !

Wir brauchen also nur die doppelseitige Anweisung zu betrachten. Diese können wir in der Form

$$\begin{array}{l}\textsf{if } b \textsf{ then } A_1\\ \textsf{elsif } \neg\, b \textsf{ then } A_2 \textsf{ else leer}\\ \textsf{end}\end{array} \tag{8.38}$$

schreiben. Da $b \vee \neg\, b$ wahr, kann die leere Nein-Alternative nie erreicht werden! Wir entnehmen (8.38), daß eine Nachbedingung Q wahr ist, wenn entweder $b \wedge \text{wp}(A_1, Q)$ oder $\neg\, b \wedge \text{wp}(A_2, Q)$ gilt. Beide Bedingungen können nicht gleichzeitig gelten, da $b \wedge \neg\, b$ falsch ist. Ist umgekehrt eine Vorbedingung P gegeben, so erhalten wir die Nachbedingung Q, wenn $\{b \wedge P\}A_1\{Q\}$ und $\{\neg\, b \wedge P\}A_2\{Q\}$ gilt. Also gilt insgesamt

Axiom der bedingten Anweisung:

$$\begin{array}{ll}\text{Aus} & \{b \wedge P\}\ A_1\ \{Q\} \text{ und } \{\neg\, b \wedge P\}\ A_2\ \{Q\}\\ \text{folgt} & \{P\}\ \textsf{if } b \textsf{ then } A_1 \textsf{ else } A_2 \textsf{ end}\ \{Q\}\end{array} \tag{8.39}$$

$$\text{wp}(\text{„}\textsf{if } b \textsf{ then } A_1 \textsf{ else } A_2 \textsf{ end}\text{“}, Q) = (b \rightarrow \text{wp}(A_1, Q)) \wedge (\neg\, b \rightarrow \text{wp}(A_2, Q)) \tag{8.40}$$

(8.40) verlangt die Berechnung von wp(A_1, Q) bzw. wp(A_2, Q) nur, wenn die entsprechende Bedingung b bzw. $\neg\, b$ wahr ist! Abgesehen davon gilt $(b \rightarrow \text{wp}(A_1, Q)) \wedge (\neg\, b \rightarrow \text{wp}(A_2, Q)) = (b \wedge \text{wp}(A_1, Q)) \vee (\neg\, b \wedge \text{wp}(A_2, Q))$.

Für die einseitige bedingte Anweisung ergibt sich hieraus

$$\begin{array}{lllll}\text{Aus } \{b \rightarrow P\}\ A\ \{Q\} \text{ und } \{\neg\, b \rightarrow Q\} & \text{folgt} & \{P\}\ \textsf{if } b \textsf{ then } A \textsf{ end}\ \{Q\}\\ \text{wp}(\text{„}\textsf{if } b \textsf{ then } A \textsf{ end}\text{“}, Q) & = & (b \rightarrow \text{wp}(A, Q)) \wedge (\neg\, b \rightarrow Q)\end{array}$$

Beispiel 8.20 (**Maximum**):

$$\begin{aligned}
&\text{wp}(\text{„if } x > y \text{ then } max := x \text{ else } max := y \text{ end“}, max = x)\\
=\ & ((x > y) \rightarrow \text{wp}(\text{„}max := x\text{“}, max = x))\\
& \quad \wedge ((x \leqslant y) \rightarrow \text{wp}(\text{„}max := y\text{“}, max = x))\\
=\ & ((x \leqslant y) \vee \text{wp}(\text{„}max := x\text{“}, max = x))\\
& \quad \wedge ((x > y) \vee \text{wp}(\text{„}max := y\text{“}, max = x))\\
=\ & (x \leqslant y \vee x = x) \wedge (x > y \vee y = x)\\
=\ & wahr \wedge (x > y \vee y = x)\\
=\ & (x \geqslant y)
\end{aligned}$$

♦

Beispiel 8.21: Unter Verwendung von Beispiel 8.18 können wir die Richtigkeit des Rumpfs der Prozedur

```
minmax(&& i,j: INT) is if i>j then h: INT := i; i := j; j := h end end
```

aus Beispiel 8.10 nachprüfen. Es gelten die Vor- und Nachbedingungen:

Vor: P: $i = i_0 \wedge j = j_0$.

Nach: Q: $i \leqslant j \wedge ((i = i_0 \wedge j = j_0) \vee (i = j_0 \wedge j = i_0))$

Gilt anfangs $i \leqslant j$, so ist das Programm richtig, da $P \wedge i \leqslant j \rightarrow Q$. Gilt aber anfangs $i > j$, so folgt aus Beispiel 8.18: $\{P \wedge i > j\}\ h : \text{INT} := i; i := j; j := h\ \{i = j_0 \wedge j = i_0 \wedge i \leqslant j\}$, also gilt auch dann die Nachbedingung Q und wir haben insgesamt

$\{P\}$ if $i > j$ then h: INT; $h := i$; $i := j$; $j := h$; end $\{Q\}$. ♦

Aufgabe 8.20: Formulieren Sie Beispiel 8.21 mit schwächsten Vorbedingungen.

Beispiel 8.22: Mit den Beispielen 8.18 und 8.21 können wir die Korrektheit des Programms zum Sortieren von 5 Zahlen in Beispiel 8.4 nachweisen. Dazu überlegen wir, daß eine Vorbedingung wie $a = a_0$ abgekürzt werden darf zu *wahr*: Selbstverständlich hat a einen Anfangswert, auch wenn wir ihn nicht kennen sollten. Die Werte aller 5 Variablen müssen eine Permutation der Anfangswerte sein. Wir kürzen diese Aussage mit perm ab. perm ist natürlich auch anfangs richtig. Damit erhalten wir das Programm 8.3 mit Beweisvorgaben. ♦

Aufgabe 8.21: Führen Sie den Beweis zu Beispiel 8.22 mit dem Kalkül der schwächsten Vorbedingungen aus. Formulieren Sie dazu ähnlich wie in den Zeilen 10, 21 zusätzlich die genauen Vorbedingungen für die Zeilen 13, 15, 24 und 26 unter Berücksichtigung der vorangehenden Bedingungen.

Beispiel 8.23: Zwei Anweisungen oder Anweisungsfolgen A, B haben die gleiche Wirkung, wenn für beliebige Vor- und Nachbedingungen P, Q gilt: Aus $\{P\}\ A\ \{Q\}$ folgt $\{P\}\ B\ \{Q\}$ und umgekehrt. A, B heißen dann **semantisch äquivalent** oder **verhaltensgleich**, in Zeichen $A \equiv B$. Die folgenden beiden bedingten Anweisungen sind äquivalent:

```
if p then                          if q then
  if q then A1 else A2 end   ≡       if p then A1 else A3 end
elsif q then A3 else A4 end        elsif p then A2 else A4 end
```

♦

Programm 8.3: Sortieren von 5 Zahlen mit Beweisvorgabe

```
{perm}                                                                          1
if a>b then h: INT; h := a; a := b; b := h end;                                 2
-- {a <= b and perm}                                                            3
if c>d then h: INT; h := c; c := d; d := h end;                                 4
-- {a <= b and c <= d and perm}                                                 5
if b>d then h: INT; h := b; b := d; d := h; h := c; a := c; c := h end;         6
-- {a <= b <= d and c <= d and perm}                                            7
if b>e                                                                          8
then if a>e then h: INT; h := e; e := d; d := b; b := a; a := h                 9
  -- {a <= b <= d <= e and c <= e and perm}                                     10
  else h: INT; h := e; e := d; d := b; b := h end;                              11
  -- {a <= b <= d <= e and c <= e and perm}                                     12
elsif d>e then h: INT; h := e; e := d; d := h end                               13
  -- {a <= b <= d <= e and c <= e and perm}                                     14
end;                                                                            15
-- {a <= b <= d <= e and c <= e and perm}                                       16
if c>b                                                                          17
then                                                                            18
  -- {a <= b <= d <= e and b < c <= e and perm}                                 19
  if c>d then h: INT; h := c: c:= d; d := h end                                 20
  -- {a <= b <= d <= e and b <= c <= d <= e and perm}                           21
elsif c>a then h: INT; h := b; b := c; c := h                                   22
-- {a <= b <= d <= e and b <= c <= d <= e and perm}                             23
else h: INT; h := a; a:= c; c := b; b := h                                      24
-- {a <= b <= d <= e and b <= c <= d <= e and perm}                             25
end                                                                             26
-- {a <= b <= c <= d <= e and perm}                                             27
```

Aufgabe 8.22: Beweisen Sie die vorstehende Äquivalenz.

Aufgabe 8.23: Zeigen Sie für beliebige Prädikate Q

$$\begin{aligned} \text{wp}(\text{„if } B \text{ then } A_1; A_2; A_3 \text{ else } A_1; A_4; A_3 \text{ end“}, Q) = \\ \text{wp}(\text{„}A_1;\ \text{if } B \text{ then } A_2 \text{ else } A_4 \text{ end};\ A_3\text{“}, Q) \\ \text{falls } B \quad = \quad \text{wp}(A_1, B) \text{ und } \neg B = \text{wp}(A_1, \neg B) \end{aligned}$$

Aufgabe 8.24: Zeigen Sie für beliebige Prädikate Q
$\text{wp}(\text{„if } B \text{ then } A_1 \text{ else } A_2 \text{ end“}, Q) = \text{wp}(\text{„if } \neg B \text{ then } A_2 \text{ else } A_1 \text{ end“}, Q)$

Aufgabe 8.25: Sind

$$\text{if } B \text{ then } A_1 \text{ else } A_2 \text{ end} \quad \text{und} \quad \text{if } B \text{ then } A_1;\ \text{if } \neg B \text{ then } A_2 \text{ end}$$

semantisch äquivalent?

Aufgabe 8.26: Die Anweisungsfolgen A' und A'' seien semantisch äquivalent. Sind dann auch

$$A_1;\ A';\ A_2 \quad \text{und} \quad A_1;\ A'';\ A_2;$$

semantisch äquivalent?

Aufgabe 8.27: Die Anweisungsfolgen A' und A'' seien semantisch äquivalent. Sind dann auch

$$\textsf{if } B \textsf{ then } A' \textsf{ else } A \textsf{ end} \quad \text{und} \quad \textsf{if } B \textsf{ then } A'' \textsf{ else } A \textsf{ end}$$

semantisch äquivalent?

8.2.5 Bewachte Anweisungen und die Fallunterscheidung

Der Kalkül der schwächsten Vorbedingungen geht auf E. W. DIJKSTRA zurück. Er führte dazu in (DIJKSTRA, 1976) bedingte Anweisungen und Schleifen mit der Technik der **bewachten Anweisungen**[27] ein. Dies ist eine indeterministische Form der Anweisungen mit der Syntax

bedingte_Anweisung ::= 'if' bewachte_Anweisungsmenge 'fi'
Schleife ::= 'do' bewachte_Anweisungsmenge 'od'
bewachte_ Anweisungsmenge ::= bewachte_Anweisung
('[]' bewachte Anweisung)*
bewachte_Anweisung ::= Wächter '→' Anweisungsfolge .

Der Wächter ist ein boolescher Ausdruck. Die bedingte Anweisung und die Schleife bestehen aus einer Menge bewachter Anweisungen in beliebiger Reihenfolge. Eine beliebige bewachte Anweisung mit wahrem Wächter wird ausgewählt und ausgeführt. Für eine Schleife wird dies wiederholt, bis alle Wächter falsch sind. Ist kein Wächter wahr, so ist die bedingte Anweisung äquivalent zu fehler, die Schleife zu leer. Die Sprachelemente sind indeterministisch, weil mehrere Wächter gleichzeitig wahr sein können.

Beispiel 8.24: Die deterministische, bedingte Anweisung if b then A_1 else A_2 end lautet in dieser Form

if $b \to A_1$
[] $\neg b \to A_2$
fi

Die while-Schleife while b loop A end lautet
do $b \to A$ od ♦

27. engl. *guarded command*

Die bewachten Anweisungen umfassen die bisherigen deterministischen Varianten. Ihr Studium liefert uns wichtige Einsichten für die Axiome der Fallunterscheidung und der (deterministischen) Schleife.

IF sei die bedingte Anweisung

$$\textsf{if } b_1 \to A_1 \ [] \ b_2 \to A_2 \ [] \cdots [] \ b_n \to A_n \ \textsf{fi}$$

Dann gilt für eine beliebige Nachbedingung R

$$\begin{aligned} \mathrm{wp}(\mathsf{IF}, R) \quad = \quad & (\exists j : 1 \leqslant j \leqslant n : b_j) \wedge \\ & (\forall j : 1 \leqslant j \leqslant n : b_j \to \mathrm{wp}(A_j, R)). \end{aligned}$$

Die Teilbedingung $(\exists j : 1 \leqslant j \leqslant n : b_j) = \bigvee_{j=1}^{n} b_j$ kürzen wir im folgenden mit BB ab. BB verlangt, daß mindestens ein b_j wahr ist; der zweite Teil der Bedingung ist die aus (8.40) bekannte Forderung, daß aus dem Zutreffen der Bedingung b_j die schwächste Vorbedingung der Anweisungsfolge A_j folgt. Insbesondere gilt $\mathrm{wp}(\mathsf{IF}, R) = \mathit{falsch}$, wenn kein b_j wahr ist; dies ist nach (8.29) die Charakterisierung der Fehleranweisung. Wir können nun $\mathrm{wp}(\mathsf{IF}, R)$ umformen:

$$\mathrm{wp}(\mathsf{IF}, R) = BB \wedge \bigwedge_{i=1}^{n} \big(b_i \to \mathrm{wp}(A_i, R)\big). \tag{8.41}$$

In dieser Form liefert $\mathrm{wp}(\mathsf{IF}, R)$ das Axiom für die Fallunterscheidung:

$$\begin{aligned} \mathsf{CASE}: \quad & \textsf{case } i \\ & \textsf{when } i_1 \textsf{ then } A_1 \\ & \vdots \\ & \textsf{when } i_n \textsf{ then } A_n \\ & \textsf{else } A_0 \\ & \textsf{end} \end{aligned}$$

ist äquivalent zur bedingten Anweisung

$$\begin{aligned} & \textsf{if} \quad i = i_1 \to A_1 \\ & [] \quad i = i_2 \to A_2 \\ & \vdots \\ & [] \quad i = i_n \to A_n \\ & [] \quad \textsf{not } (i = i_1) \wedge \textsf{not } (i = i_2) \wedge \cdots \wedge \textsf{not } (i = i_n) \to A_0 \\ & \textsf{fi} \end{aligned}$$

wenn die i_j paarweise verschieden sind. Es ist immer genau eine der Vorbedingungen, also auch BB wahr. Daher können wir diesen Term in (8.41) weglassen und erhalten mit $BB' = \bigvee_{j=1}^{n} (i = i_j)$:

Axiom der Fallunterscheidung:

$$\begin{array}{ll} \text{Aus} & \forall j : 1 \leqslant j \leqslant n \{ i = i_j \wedge P \}\ A_j\ \{Q\} \\ \text{und} & \{ \neg BB' \wedge P \}\ A_0\ \{Q\} \\ \text{folgt} & \{P\}\ \mathsf{CASE}\ \{Q\}, \end{array}$$

$$\text{wp}(\mathsf{CASE}, Q) = (\neg BB' \rightarrow \text{wp}(A_0, Q)) \wedge \bigwedge_{j=1}^{n} (i = i_j \rightarrow \text{wp}(A_j, Q)).$$

Das Fehlen der Alternative „else A_0“ ist nach den Regeln der meisten imperativen Sprachen als „else leer“ und nicht als „else fehler“ zu interpretieren. Eine solche Fallunterscheidung ist daher nur dann sinnvoll, wenn

$$\begin{aligned} \neg BB' \rightarrow \text{wp}(\mathsf{leer}, Q) &= \neg BB' \rightarrow Q \\ &= BB' \vee Q \\ &= \neg Q \rightarrow BB' \end{aligned}$$

wahr ist: Bei einer Fallunterscheidung ohne else-Alternative muß die Nachbedingung Q bereits vor Ausführung erfüllt sein, wenn keiner der Fälle $i = i_j$ vorliegt. Der Programmierer muß dafür sorgen, daß alle Alternativen, die eine Aktion erfordern, auch berücksichtigt sind.

8.2.6 Schleifen

Wenn wir in den Axiomen für die bedingte Anweisung oder die Fallunterscheidung die Ausführung einer Anweisung $\{P\}\ A\ \{Q\}$ auf das Ziehen eines logischen Schlusses $P \rightarrow Q$ reduzieren, entsteht bemerkenswerterweise eine Vorschrift, um einen mathematischen Beweis durch Fallunterscheidung zu führen. Aus Vor- und Nachbedingung werden Voraussetzung und Behauptung. Die Schwierigkeiten bei einer Fallunterscheidung ohne else-Alternative entsprechen dem Problem des Nachweises, daß die betrachteten Fälle die Behauptung vollständig abdecken.

In gleicher Weise entsprechen die nachfolgenden Axiome für Schleifen dem Beweisprinzip der vollständigen Induktion: Ist $P(k)$ die Aussage, daß P für alle $j < k$ gilt, so ist im Induktionsschluß die Gültigkeit von $P(k + 1)$ zu beweisen. Ersetzen wir den Induktionsschluß durch Ausführung des Schleifenrumpfes, so sehen wir, daß der Schleifenrumpf nacheinander Zusicherungen $P(0), P(1), P(2), \ldots$ herstellen muß. Diese dienen sowohl als Vor- wie als Nachbedingung des Schleifenrumpfes. Können wir die $P(k)$ ohne Bezug auf k als Bedingung P formulieren, so ist P die uns bereits bekannte **Schleifeninvariante**.

Da wir unendliche Mengen algorithmisch nicht verarbeiten können, müssen wir im Unterschied zur mathematischen Induktion zusätzlich das Terminieren

der Schleife zeigen. An dieser Stelle zeigt sich der Unterschied des wp-Kalküls und der HOARE-Logik.

Wir betrachten eine Schleife

$$\mathsf{DO}: \ \mathsf{do}\ b_1 \to A_1\ []\ b_2 \to A_2\ [] \cdots []\ b_n \to A_n\ \mathsf{od}$$

IF bezeichne die entsprechende bedingte Anweisung mit do ··· od ersetzt durch if ··· fi. R sei die Nachbedingung von DO. Wenn die Schleife 0-mal durchlaufen wird, darf kein Wächter wahr sein und die Nachbedingung muß bereits gelten. Wir haben also die schwächste Vorbedingung

$$\begin{aligned} H_0(R) &= R \wedge \neg(\exists j : 1 \leqslant j \leqslant n : b_j) \\ &= R \wedge \bigwedge_{j=1}^{n} \neg b_j. \end{aligned}$$

Für höchstens k-maliges Durchlaufen der Schleife ergibt sich die Bedingung

$$H_k(R) = \mathrm{wp}\left(\mathsf{IF}, H_{k-1}(R)\right) \vee H_0(R). \tag{8.42}$$

Diese Bedingung gibt unser intuitives Verständnis (8.13) einer Schleife wieder: Wenn ein Wächter wahr ist, dann führe die IF-Anweisung aus und wiederhole den Vorgang. Wir können das jetzt beweisen:

Satz 8.4: *Die Anweisungen*

while b loop A end

und

if b then A; while b loop A end end

sind semantisch äquivalent.

Beweis: Wegen (8.42) gilt bei höchstens k-maligen Durchlaufen der Schleife

$$\{H_k(R)\}\ \mathsf{if}\ b\ \mathsf{then}\ A;\ \{H_{k-1}(R)\}\ \mathsf{while}\ b\ \mathsf{loop}\ A\ \mathsf{end}\ \mathsf{end}\ \{R \wedge \neg b\}. \tag{8.43}$$

$H_k(R)$ ist die schwächste Vorbedingung; für jede andere Vorbedingung P muß $P \to H_k(R)$ gelten. Durch vollständige Induktion sieht man, daß auch

$$\{H_k(R)\}\ \mathsf{while}\ b\ \mathsf{loop}\ A\ \mathsf{end}\ \{R \wedge \neg b\}$$

gilt: Das Präfix „if b then A“ in (8.43) entspricht dem Übergang $(k-1) \to k$. ♦

Im Beweis mußten wir annehmen, daß die Schleife höchstens k-mal durchlaufen wird. Wir fordern nun, daß ein solches k existiert und erhalten

Schleifenaxiom mit schwächster Vorbedingung:

$$\mathrm{wp}(\mathsf{DO}, R) \quad = \quad \exists k \geqslant 0 : H_k(R). \tag{8.44}$$

Für die sequentielle while-Schleife

$$\text{WHILE}: \ \textsf{while}\ b\ \textsf{loop}\ A\ \textsf{end}$$

ergibt sich hieraus:

Schleifenaxiom für while-Schleife mit schwächster Vorbedingung:

$$\begin{aligned} H_0(R) &= R \wedge \neg b, \qquad (8.45)\\ H_k(R) &= \text{wp}(\text{„}\textsf{if}\ b\ \textsf{then}\ A\ \textsf{end}\text{“}, H_{k-1}) \vee R \wedge \neg b,\\ \text{wp}(\text{WHILE}, R) &= \exists k \geqslant 0 : H_k(R). \end{aligned}$$

Dieses Axiom ist wegen der Folge $H_0(R), H_1(R), \ldots$ verschiedener Bedingungen schwierig zu handhaben. Wir benötigen es gerade bei den einfachsten Schleifen, nämlich Zählschleifen, die eine Folge von n Eingabedaten verarbeiten. Die Vorbedingung $H_k(R)$ hat dann die allgemeine Form „k Datensätze erfolgreich bearbeitet“.

Es gilt nun der

Satz 8.5 (**Schleifeninvariante**): *Sei* DO *eine Schleife und* IF *die entsprechende bedingte Anweisung,* $BB = \bigvee_{j=1}^{n} b_j$, *und* P *eine Zusicherung mit*

$$P \wedge BB \rightarrow \text{wp}(\text{IF}, P). \qquad (8.46)$$

Dann gilt $P \wedge \text{wp}(\text{DO}, \textit{wahr}) \rightarrow \text{wp}(\text{DO}, P \wedge \neg BB)$.

Für die while-Schleife lautet dieser Satz

Korollar 8.6: *Falls* $P \wedge b \rightarrow \text{wp}(\text{WHILE}, P)$, *dann*

$$P \wedge \text{wp}(\text{WHILE}, \textit{wahr}) \rightarrow \text{wp}(\text{WHILE}, P \wedge \neg b). \qquad (8.47)$$

Jede Zusicherung P, die dem Satz bzw. dem Korollar genügt, ist eine Schleifeninvariante der betreffenden Schleife.

Beweis von Satz 8.5: Die Bedingung wp(DO, *wahr*) entspricht der Forderung, daß die Schleife terminiert: da jeder erreichbare Zustand z der trivialen Bedingung *wahr* genügt, reduziert sich wp(DO, *wahr*) nach (8.44) auf die Forderung, daß ein k existiert, so daß nach k-maliger Wiederholung $\neg BB$ gilt.

Mit den Bezeichnungen des Satzes haben wir

$$\begin{aligned} H_0(\textit{wahr}) &= \neg BB\\ H_k(\textit{wahr}) &= \text{wp}(\text{IF}, H_{k-1}(\textit{wahr})) \vee \neg BB \qquad (8.48)\\ H_0(P \wedge \neg BB) &= P \wedge \neg BB\\ H_k(P \wedge \neg BB) &= \text{wp}(\text{IF}, H_{k-1}(P \wedge \neg BB)) \vee (P \wedge \neg BB) \qquad (8.49) \end{aligned}$$

Durch vollständige Induktion ergibt sich

$$P \wedge H_k(wahr) \rightarrow H_k(P \wedge \neg BB)$$

Für $k = 0$ ist dies nach Definition richtig. Für $k > 0$ haben wir

$$\begin{array}{rcl} P \wedge H_k(wahr) & = & P \wedge (\mathrm{wp}(\mathsf{IF}, H_{k-1}(wahr)) \vee \neg BB) \\ & = & (P \wedge BB \wedge \mathrm{wp}(\mathsf{IF}, H_{k-1}(wahr))) \vee P \wedge \neg BB \\ & \rightarrow & \mathrm{wp}(\mathsf{IF}, P) \wedge \mathrm{wp}(\mathsf{IF}, H_{k-1}(wahr)) \vee P \wedge \neg BB \\ & = & \mathrm{wp}(\mathsf{IF}, P \wedge H_{k-1}(wahr)) \vee P \wedge \neg BB \\ & \rightarrow & \mathrm{wp}(\mathsf{IF}, H_{k-1}(P \wedge \neg BB)) \vee P \wedge \neg BB \\ & = & H_k(P \wedge \neg BB) \end{array}$$

Dabei haben wir nacheinander (8.48), die Feststellung $\mathrm{wp}(\mathsf{IF}, R) \rightarrow BB$, die Voraussetzung (8.46) des Satzes, die Identität (8.21), die Induktionsvoraussetzung für $k - 1$ und (8.49) benutzt.

Schließlich erhalten wir

$$\begin{array}{rcl} P \wedge \mathrm{wp}(\mathsf{DO}, wahr) & = & \exists k \geqslant 0 : P \wedge H_k(wahr) \\ & \rightarrow & \exists k \geqslant 0 : H_k(P \wedge \neg BB) \\ & = & \mathrm{wp}(\mathsf{DO}, P \wedge \neg BB). \end{array}$$ ♦

Axiom der partiellen Korrektheit einer Schleife:

$$\text{Aus } \{P \wedge b\}\ A\ \{P\} \quad \text{folgt} \quad P\ \{\mathsf{while}\ b\ \mathsf{loop}\ A\ \mathsf{end}\}\ P \wedge \neg b \quad (8.50)$$

Hier haben wir die Voraussetzung wp(„while b loop A end“, $wahr$) von Korollar 8.6, die die Terminierung garantiert, weggelassen. Die Terminierung überprüfen wir getrennt mit einer Terminierungsfunktion $t(k)$.

Beispiel 8.25 (**Lineare Suche**): Gegeben sei eine vorbesetzte Reihung $a[0 : n-1]$ und ein Wert x, der unter den $a[j]$ vorkommt. Wir wollen den kleinsten Index i mit $x = a[i]$ berechnen. Die Spezifikation dieser Aufgabe lautet

$$\begin{array}{ll} \text{Vor:} & Q\colon 0 < n \wedge x \in a[0 : n-1] \\ \text{Nach:} & R\colon 0 \leqslant i < n \wedge x \notin a[0 : i-1] \wedge x = a[i] \end{array}$$

Es gilt $R : 0 \leqslant i < n \wedge (\forall j\colon 0 \leqslant j < i\colon x \neq a[j]) \wedge x = a[i]$, wenn wir $x \notin a[0 : i-1]$ ausschreiben.

Eine Schleifeninvariante erhält man aus der Nachbedingung: Wegen des Schleifenaxioms (8.50) müßte sie die Form $R\colon P \wedge \neg b$ haben, wobei der Wächter b seinen Wert während des Schleifenablaufs ändern muß. Wir erreichen das mit $P\colon 0 \leqslant i < n \wedge (\forall j\colon 0 \leqslant j < i\colon x \neq a[j])$, $b\colon x \neq a[j]$. Dabei haben wir mit j einen Index in b eingeführt, der die Veränderlichkeit von b und für $j = i$ die ursprüngliche Bedingung herstellt. Wir sehen, daß b auch in P vorkommt.

Ferner überlegen wir, daß wir eine Invariante der Form $\forall j\colon 0 \leqslant j < i\colon \cdots$ mit einer Schleife $j := 0$; while $\cdots$ loop $j := j + 1$ end erhalten können, da aus dem Zuweisungsaxiom unmittelbar $\{0 \leqslant j < i\}\ j := j + 1\ \{0 \leqslant j < i \vee j = i\}$ folgt. Setzen wir nun noch b ein, so erhalten wir das Programmstück

```
j := 0;
while x /= a[j] loop {P} j := j + 1 end
```

mit Schleifeninvariante P.[28] ♦

Beispiel 8.26 (**Summierung**): Gegeben sei eine vorbesetzte Reihung $a[0 : n-1]$ ganzer Zahlen. Wir wollen die Summe der Reihungselemente berechnen. Es gilt:

Gegeben: a : ARR$[n]$(INT);

Gesucht: $s := \sum_{j=0}^{n-1} a_j$

Als Vor- bzw. Nachbedingung erhalten wir:

$$\begin{aligned} Q\colon &\quad \mathit{wahr} \\ R\colon &\quad s = \sum_{j=0}^{n-1} a[j] \end{aligned}$$

Um hier die Nachbedingung in die Form $P \wedge \neg b$ zu bringen, setzen wir statt der Obergrenze n eine Variable i ein und benutzen $P\colon 0 \leqslant i \leqslant n \wedge s = \sum_{j=0}^{i-1} a[j]$, $b : i \neq n$. In Verallgemeinerung des vorigen Beispiels können wir mit dem Schleifenrumpf $s := s + a[i];\ i := i + 1$ von i zu $i + 1$ übergehen. Für $i = 0$ müssen wir $s = \sum_{j=0}^{-1} a[j] = 0$ setzen. Damit erhalten wir das Programm

```
{wahr}
i := 0; s := 0;
while i /= n loop {P} s := s + a[i]; i := i + 1 end
{R}
```

♦

Beispiel 8.27 (**Telefonbuchproblem, Gries**): Wir wollen finden, wie oft der häufigste Namen im örtlichen Telefonbuch vorkommt. Das Telefonbuch ist alphabetisch nach diesen Namen geordnet. Da uns der Rest eines Telefonbucheintrags und die Länge der Namen nicht interessieren, beschreiben wir die Aufgabe abstrakt so: Gegeben sei eine geordnete Reihung $a[0 : n-1]$, $n \geqslant 1$. Gesucht ist die maximale Häufigkeit k eines Elements x, $x \in a$.

Wir wissen:

- $a[0 : n-1]$ enthält k gleiche Elemente und k$\geqslant 1$.

28. Hier wie im folgenden ist $\{\cdots\}$ oder ein Kommentar unmittelbar nach dem Wortsymbol loop stets eine Schleifeninvariante.

- $a[0 : n-1]$ enthält keine $k+1$ gleichen Elemente.
- Die k gleichen Elemente folgen in einem Abschnitt $a[p : p+k-1]$, $p+k-1 \leqslant n-1$, aufeinander.

Dies liefert die Nachbedingung.

R_n: $\exists p: 0 \leqslant p < n-k \wedge a[p] = a[p+k-1]) \wedge (\forall j: 0 \leqslant j < n-k-1:$
$a[j] \neq a[j+k])$.

Die vorigen Beispiele legen nahe, R_i für $i = 1, \ldots, n$ als Schleifeninvariante und b: $i < n$ als Schleifenbedingung zu benutzen. Beim Übergang $i \to i+1$ kann sich k nur erhöhen, wenn der letzte Term $\forall j: 0 \leqslant j < i-k-1: a[j] \neq a[j+k]$ der Invariante verletzt ist. Wenn dies zuvor nicht der Fall war, muß $a[i] = a[i-k]$ gelten. Dies führt zu der Lösung

```
i := 1; k := 1;
while i<n
loop {R_i}
  if a[i]=a[i-k] then
    k := k + 1;
  end;
  i := i+1
end;
{R}
```

oder mit einer Zählschleife

```
k := 1;
loop
  constant i: INT := 1.upto!(n-1);
  if a[i]=a[i-k] then
    k := k + 1;
  end
end
```

Aus $a[i] \neq a[i-1]$ folgt, daß $a[i] = a[i-k]$ erst wieder möglich ist, wenn i um k erhöht wurde. Wir können daher die erste Fassung durch das endgültige Programm 8.4 ersetzen. Während der Aufwand der ersten Fassung immer $O(n)$

Programm 8.4: Längste Folge gleicher Namen

```
i := 1; k := 1;
while i<n
loop
  if a[i]=a[i-k] then
    k := k + 1; i := i+1
  elsif a[i] /= a[i-1] then i := i+k
  else i := i+1
  end;
end;
```

ist, erhalten wir jetzt in vielen Fällen einen sublinearen Aufwand. ♦

Aufgabe 8.28: Erweitern Sie das Telefonbuch-Beispiel so, daß es den ersten bzw. letzten Namen mit Häufigkeit k liefert.

Beispiel 8.28 (**Hoare**): Für späteren Gebrauch entwickeln wir eine Prozedur

zerlege(&& a: ARR[∗](INT); m, n: INT; &i, j: INT),

die aus der Vorbedingung

$$P\colon 0 \leqslant m \leqslant n < a.asize \tag{8.51}$$

die Nachbedingung

$$Q\colon (j < i) \wedge \mathsf{perm}(a) \wedge \forall p, q\colon \big((m \leqslant p < i) \wedge (j < q \leqslant n) \rightarrow (a[p] \leqslant a[q])\big) \tag{8.52}$$

folgert. perm(a) verlangt, daß die Reihung a gegenüber der Vorbesetzung höchstens permutiert worden ist.

Zur Lösung der Aufgabe könnten wir das Minimum aller $a[k]$, $m \leqslant k \leqslant n$, bestimmen. Hat dieses den Index k_0, so vertauschen wir die Werte von $a[m]$ und $a[k_0]$ und setzen $j = m$, $i = m + 1$. Damit ist die Nachbedingung erfüllt.

Eine allgemeinere Lösung geht von einem beliebigen Wert r mit $\min a[k] \leqslant r \leqslant \max a[k]$, $m \leqslant k \leqslant n$, z. B. $r = a[(m + n)\ \mathsf{div}\ 2]$, aus und bestimmt i, j so, daß

$$(m \leqslant p < i) \quad \rightarrow \quad a[p] \leqslant r, \tag{8.53}$$

$$(j < q \leqslant n) \quad \rightarrow \quad r \leqslant a[q], \tag{8.54}$$

$$(j < p < i) \quad \rightarrow \quad a[p] = r, \tag{8.55}$$

wobei (8.55) für den Fall $j < i$ aus den vorangehenden Bedingungen folgt. Die Bedingungen (8.53) – (8.55) definieren die Zerlegung der Indexbereiche wie in Abb. 8.6, die der Prozedur den Namen gibt, während die ursprüngliche Nach-

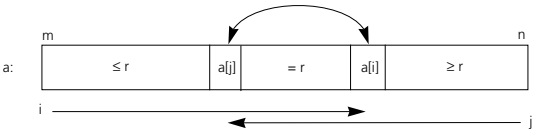

Abbildung 8.6: Schema der Zerlegung

bedingung nur eine Aufteilung in zwei überlappende Indexbereiche vorsieht.

Die Zusicherung

$$Inv\colon \forall p, q\colon \big((m \leqslant p < i) \wedge (j < q \leqslant n) \rightarrow (a[p] \leqslant a[q])\big), \tag{8.56}$$

also der zweite Teil der Nachbedingung Q, ist erfüllt, wenn wir $i = m, j = n$ wählen. Daher liegt es nahe, die Aufgabe durch eine Schleife mit Schleifeninvariante *Inv* und Endebedingung $j < i$ zu lösen. Die Prozedur

```
zerlege(&& a: ARR[*](INT); m,n: INT; & i,j: INT) is
-- Vor:  0 <= m <= n < a.asize
-- Nach: Q
  r: INT := a[(m+n) div 2];
  i := m;
  j := n;
  while i <= j
  loop -- Schleifeninvariante Inv
  ...
  end;
end; -- zerlege
```

löst das Problem, wenn wir den Schleifenrumpf passend ausfüllen können.

Dazu überlegen wir, daß wir i erhöhen und j erniedrigen dürfen, solange $a[i] < r < a[j]$ gilt, und daher die Schleifenbedingung erhalten bleibt. Dies erreichen wir mit

```
while a[i] < r loop i := i + 1 end;
while a[j] > r loop j := j − 1 end;
```
(8.57)

Danach gilt

$$a[i] \geqslant r \geqslant a[j]. \tag{8.58}$$

Wenn nun die Endebedingung $j < i$ noch nicht erfüllt ist, vertauschen wir $a[i]$ mit $a[j]$; danach gilt die Schleifeninvariante sogar einschließlich der Indizes i, j, die wir daher noch um 1 erhöhen bzw. erniedrigen dürfen, ohne *Inv* zu verletzen. Dies ergibt zusammen den Schleifenrumpf

```
while a[i]<r loop i := i+1 end;
while a[j]>r loop j := j-1 end;
if i <=j
then h: INT := a[i]; a[i] := a[j]; a[j] := h;
    i := i+1; j :=j -1
end
```

In jedem Schleifendurchlauf ändert sich mindestens einer der Werte i, j: Entweder geschieht dies in einer der beiden while-Schleifen; oder die Bedingung $i \leqslant j$ gilt, und beide Indizes ändern sich. Wegen $m \leqslant i \leqslant j \leqslant n$ gibt es aber maximal $2(n - m)$ Änderungen der Indizes und daher auch maximal $2(n - m)$ Schleifendurchläufe: $t(i, j) = 2n - i - j$ ist eine Terminierungsfunktion für die Schleife. Wegen $Q = Inv \wedge (i > j)$ ist somit unsere Prozedur korrekt. Für den Aufwand erhalten wir

$$T_{\text{zerlege}}(m, n) \leqslant 2(n - m) = O(n - m). \qquad \blacklozenge \tag{8.59}$$

Aufgabe 8.29: Formulieren Sie eine Prozedur, die die Zerlegung mit der im Beispiel zuerst erwähnten Minimumbestimmung löst, und beweisen Sie deren Korrektheit.

Aufgabe 8.30: Formulieren Sie Invarianten für die Schleifen (8.57) so, daß anschließend die Zusicherung (8.58) gilt. Wieso terminieren diese Schleifen?

Aufgabe 8.31: Wie groß ist die Anzahl der Vergleiche in der Prozedur zerlege?

Die Axiome für bewachte Schleifen haben eine interessante Fortsetzung für parallele Programme, die auf (Owicki und Gries, 1976) zurückgeht und die wir hier nur andeuten können, vgl. auch (Apt und Olderog, 1997).

Wir betrachten eine Schleife do $b_1 \to A_1$ [] $b_2 \to A_2$ od mit der Eigenschaft, daß $b_i, i = 1, 2$ anfangs beide wahr sind und falsch werden, sobald A_i ausgeführt wurde. Die Schleife wird also genau zweimal ausgeführt. Nach den Regeln der bewachten Anweisung können die A_i in beliebiger Reihenfolge ausgeführt werden, zuerst A_1 dann A_2 oder umgekehrt. Wir fragen, was wir über das Ergebnis wissen, wenn A_1, A_2 nicht nur nacheinander, sondern auch zeitlich verzahnt oder echt parallel ausgeführt werden können. Wir schreiben sie dann in der Form par $A_1 \parallel A_2$ end und verzichten auf die Wächter, deren Eigenschaften wir ja festgelegt haben. Gilt $\{P_i\}\, A_i\, \{Q\}, i = 1, 2$, so möchten wir eigentlich schließen:

$$\text{Aus } \{P_i\}\, A_i\, \{Q_i\}, i = 1, 2 \text{ folgt } \{P_1 \wedge P_2\}\ \textsf{par}\ A_1 \parallel A_2\ \textsf{end}\ \{Q_1 \wedge Q_2\} \tag{8.60}$$

Wegen Satz 8.5 können wir (8.60) nicht einmal bei Hintereinanderschaltung der A_i ohne Zusatzbedingung erwarten: Wird nämlich zuerst A_1 ausgeführt, so muß danach immer noch P_2 gelten, um Q_2 mit der Ausführung von A_2 zu erreichen. Ferner darf A_2 die Aussage Q_1 nicht zerstören.

Werden A_1, A_2 in Zustandsübergänge $\{P_{ij}\}\, A_{ij}\, \{P_{ij+1}\}, i = 1, 2, j = 0, \ldots, n-1, n = n(i)$, aufgespalten, die parallel oder verzahnt, aber unter Wahrung der Reihenfolge für jedes i ausgeführt werden, so erhalten wir für die A_{ij} die gleiche Nebenbedingung.

Owicki und Gries nennen A_1, A_2 **interferenzfrei**, wenn man den Beweis von $\{P_i\}\, A_i\, \{Q_i\}$ so führen kann, daß die gleichzeitige Ausführung der anderen Anweisung den Beweis nicht stört. Dies ist offenbar eine schärfere Formulierung der Nachbedingung. Sie konnten dann zeigen

Satz 8.7 (**Owicki,Gries**): *Es gilt (8.60), wenn die Anweisungen A_1, A_2 interferenzfrei sind.*

Der Satz läßt sich auf die Parallelausführung von n Anweisungen $A_1, \ldots, A_n$ verallgemeinern.

Beispiel 8.29: Wir betrachten die Zuweisung

$$\{x = x_0\}\ \textsf{par}\ x := x + a \parallel x := x + b\ \textsf{end}\ \{x = x_0 + a + b\} \tag{8.61}$$

Wenn wir die Einzelzusicherungen in der Form $\{x = x_0\}\ x := x + a\ \{x = x_0 + a\}$ und $\{x = x_0\}\ x := x + b\ \{x = x_0 + b\}$ formulieren, können wir das gewünschte Ergebnis nicht erreichen. Schreiben wir aber

$$\begin{array}{lll} \{x = x_0 \vee x = x_0 + b\} & x := x + a & \{x = x_0 + a \vee x = x_0 + b + a\} \\ \{x = x_0 \vee x = x_0 + a\} & x := x + b & \{x = x_0 + b \vee x = x_0 + a + b\} \end{array}$$

so stört die jeweils andere Anweisung den Beweis nicht, wenn wir annehmen, daß die beiden Zuweisungen atomar sind, d. h. daß sich nicht beide Anweisungen den Wert von x gleichzeitig holen können und unabhängig voneinander zurückschreiben. Wir erhalten dann

$$\begin{array}{c} \left\{ (x = x_0 \vee x = x_0 + b) \wedge (x = x_0 \vee x = x_0 + a) \right\} \\ \textsf{par}\ x := x + a \parallel x := x + b\ \textsf{end} \\ \left\{ (x = x_0 + a \vee x = x_0 + a + b) \wedge (x = x_0 + b \vee x = x_0 + a + b) \right\} \end{array}$$

Für den Beweis können wir $a, b \neq 0$ und $a \neq b$ voraussetzen. Dann gilt $x = x_0 + \alpha \wedge x = x_0 + \beta = \mathit{falsch}$ für alle Kombinationen $\alpha \neq \beta$, da die beiden Vergleiche in den Konjunktionen nicht gleichzeitig wahr sein können. Daher haben wir

$$\begin{aligned}
&(x = x_0 \vee x = x_0 + b) \wedge (x = x_0 \vee x = x_0 + a) \\
&\quad = \quad (x = x_0) \vee (x = x_0 \wedge x = x_0 + a) \vee \\
&\qquad\quad (x = x_0 + b \wedge x = x_0) \vee (x = x_0 + b \wedge x = x_0 + a) \\
&\quad = \quad (x = x_0) \vee \mathit{falsch} \vee \mathit{falsch} \vee \mathit{falsch} \\
&\quad = \quad (x = x_0)
\end{aligned}$$

Analog erhält man

$$\begin{aligned}
(x_0 = x_0 + a \vee x_0 = x_0 + a + b) \wedge (x_0 = x_0 + b \vee x_0 = x_0 + a + b) \\
= \quad (x = x_0 + a + b)
\end{aligned}$$

Das Programm (8.61) ist also korrekt, wenn die beiden Zuweisungen atomar sind. ♦

Die Voraussetzung, daß gewisse Zuweisungen $x := a$ **atomar** oder unteilbar sein müssen, d. h. als ein Zustandsübergang ablaufen, während dessen die andere Anweisung nicht auf die Variable x zugreifen darf, ist charakteristisch für parallele Programme mit gemeinsamen Variablen. Oft muß man größere Programmteile als atomar kennzeichnen; dies läßt sich technisch jedoch immer auf die Unteilbarkeit einer bedingten Zuweisung if $a \neq w$ then $a := w$ end zurückführen, vgl. auch Bd. I, Abschnitt 2.5.

8.2.7 Prozeduren

Wir beschränken uns zunächst auf eigentliche Prozeduren, die höchstens einen Eingabe- und einen Ausgabeparameter haben. Diese werden mit Wert- bzw. Ergebnisaufruf übergeben. Ferner nehmen wir an, daß Nebenwirkungen nur durch explizite Zuweisungen auftreten können; daher schreiben wir auch die in den Abschnitten 8.1.4 bzw. 8.2.2 eingeführte Nebenbedingung zulässig(*ausdruck*) bei der Berechnung von Ausdrücken nicht explizit.

Gegeben sei eine Prozedurvereinbarung

$$\begin{aligned}
&p(x\colon T; \&y\colon T') \\
&-- \ \mathit{Vor}\colon \quad P \\
&-- \ \mathit{Nach}\colon Q \\
&\textsf{is } A \textsf{ end}
\end{aligned} \tag{8.62}$$

mit Aufrufen $p(a, \&b)$. Wir setzen voraus, daß $\{P\}\ A\ \{Q\}$ gilt.

P heißt eine Vorbedingung, Q eine Nachbedingung der Prozedur. Die Vorbedingung P enthält den Ergebnisparameter y nicht, da sein Wert noch undefiniert ist. Die Nachbedingung Q beschreibt den Wert y, sowie etwaige weitere Nebenwirkungen der Prozedur auf globale Größen als Funktion des Wertes x_0 des Eingabeparameters zu Beginn der Prozedur und der Eingangswerte etwa verwendeter globaler Größen. Wir sprechen im folgenden vom Wert des Eingabeparameters x, wenn wir seinen Anfangswert meinen.

Wie in Abschnitt 8.1.6.4 bemerkt, besteht die von außen sichtbare Zustandsänderung eines Aufrufs $p(a, \&b)$ aus der Wirkung einer oder mehrerer Zuweisungen

$$y := w_0; v_1 := w_1; ...; v_k = w_k \tag{8.63}$$

an den Ergebnisparameter y und an Variable v_i, die auf der Aufrufseite bekannt sind. Im einfachsten Fall ist $k = 0$ und es gibt nur die Zuweisung an y, die bei Ausführung durch eine Zuweisung an b ersetzt wird. Für $k \geqslant 1$ können die Variablen v_i nicht im Prozedurrumpf A vereinbart sein und es kann sich auch nicht um den Parameter x handeln, da Änderungen dieser Größen nicht außerhalb der Prozedur sichtbar sind; solche v_i können also nur globale Variable sein, auf deren Wert der Aufruf $p(a, \&b)$ eine Nebenwirkung ausübt. Wir setzen voraus, daß solche globalen Variablen an der Aufrufstelle die gleiche Bedeutung haben wie in der Prozedurvereinbarung. Die Bezeichner x, y der Parameter seien im Kontext des Prozeduraufrufs nicht definiert. Diese Bedingungen können wir durch geeignete Ersetzung von Bezeichnern durch neue, im Programm noch nicht vorkommende Bezeichner immer gewährleisten.

Die zugewiesenen Werte w_i können Konstante sein oder vom Wert des Eingabeparameters x abhängen, der beim Aufruf durch das aktuelle Argument a ersetzt wird, oder sie könnten zusätzlich von Werten globaler Größen g_j abhängen, auf die der Prozedurrumpf zugreift.

Somit kann die Vorbedingung P nur Aussagen über x und die verwandten globalen Größen g_j enthalten; die Nachbedingung Q beschränkt sich auf Aussagen über die Werte von y und der globalen Variablen v_i, an die zugewiesen wird. Sollten P oder Q weitere Teilaussagen T enthalten, so wird deren Gültigkeit durch den Aufruf der Prozedur nicht verändert; die Teilaussagen müssen also sowohl in P als auch in Q vorkommen.

Der Prozedurrumpf A kann bedingte Anweisungen, Schleifen und weitere Prozeduraufrufe enthalten. Auch kann er an den Ergebnisparameter y oder eine der Variablen v_i aus (8.63) mehrfach zuweisen. Die Zuweisung an eine Variable v_i könnte auch unter Bedingung stehen und bei Nichterfüllung entfallen. Wenn wir die von außen beobachtbare Wirkung eines Prozeduraufrufs auf die Wirkung der Zuweisungen (8.63) reduzieren, abstrahieren wir von allen diesen Einzelheiten, die den Aufrufer der Prozedur ja auch nichts angehen. Insbesondere zählt bei Mehrfachzuweisung an y oder ein v_i immer nur die letzte Zuweisung zu (8.63).

In der Programmiersprache EIFFEL kann man Vor- und Nachbedingungen P, Q nicht nur als Kommentare, sondern als Teil des Kopfs einer Prozedurvereinbarung mit der Interpretation

```
p(x: T, &y: T')
        is assert P;
          begin A end;
          assert Q;
      end
```
(8.64)

schreiben. In den meisten anderen Programmiersprachen müssen diese Fehlerabfragen explizit programmiert werden.

Wir geben hier die Vor- und Nachbedingungen als Kommentare an, um auch Aussagen, die allgemeiner sind als boolesche Ausdrücke, formulieren zu können. Dem Leser wird empfohlen, sich dieses Verfahren als ständigen Programmierstil anzugewöhnen. Man erleichtert damit den Schreibern von Prozeduraufrufen das Leben. Sie kennen die Wirkung des Prozedurrumpfes, ohne die ganze Prozedur studieren zu müssen. Die assert-Anweisung sollte zusätzlich benutzt werden, um explizite und mit booleschen Ausdrücken darstellbare Bedingungen, insbesondere Bereichsangaben wie $a > 0$, $1 \leqslant b \leqslant 5$ usw. zu prüfen.

Wir untersuchen nun den Zusammenhang zwischen den Vor- und Nachbedingungen P, Q und den Zusicherungen R, S eines Aufrufs $\{R\}p(a, \&b)\{S\}$.

Bei einer parameterlosen Prozedur sind P, Q zugleich Zusicherungen für den Prozeduraufruf p: Für einen Aufruf p von p is A end mit Vorbedingung P und Nachbedingung Q gilt $\{P\}\ p\ \{Q\}$. Aus $\mathrm{wp}(\text{„}A\text{“}, Q) = P$ folgt $\mathrm{wp}(\text{„}p\text{“}, Q) = P$. Daher muß für R, S gelten $R \to P$ und $Q \to S$.

Bei einer Prozedur mit Parametern wie (8.62) ohne weitere Nebenwirkungen, $k = 0$ in (8.63), können wir die Wirkung des Rumpfes A durch die Zuweisung $y := w_0$ ersetzen. Dem geht bei einem Aufruf $p(a, \&b)$ die Zuweisung $x := a$ des Arguments an den Eingabeparameter voraus und es folgt die Zuweisung $b := y$ des Ergebnisses an das zugehörige Argument. Der Aufruf wird also abstrahiert zu der Zuweisungsfolge

$$\begin{aligned} &x := a;\\ &y := w_0;\\ &b := y \end{aligned} \tag{8.65}$$

Die Beweisvorgabe hierfür lautet

$$\{R\}x := a;\ \{P\}\ y := w_0;\ \{Q\}\ b := y\ \{S\}\ . \tag{8.66}$$

Wir könnten nun den Beweis von (8.66) für jeden Prozeduraufruf $p(a, \&b)$ einzeln führen. Wenn wir aber den Korrektheitsbeweis $\{P\}\ A\ \{Q\}$ für den Prozedurrumpf A wieder verwenden wollen, muß gemäß dem Zuweisungsaxiom, angewandt auf $x := a$ und $b := y$ gelten: $R \to P[a/x]$ und $Q \to S[y/b]$. Für den eigentlichen Prozedurrumpf, abstrahiert zu $y := w_0$, erhalten wir $P \to Q[w_0/y]$. Diese Folgerungen müssen für alle Werte w_0 gelten, die der Prozedurrumpf an y zuweisen könnte. Mit $S[y/b, w_0/y] = S[w_0/b]$ erhalten wir folglich im Kalkül der schwächsten Vorbedingungen für R in der Beweisvorgabe (8.66) die Bedingung

$$PR\colon P[a/x] \wedge \forall w\colon (Q[w/y] \to S[w/b]) \tag{8.67}$$

als Vorbedingung für den Prozeduraufruf $p(a, \&b)$. Es gilt also

Satz 8.8 (**Axiom für Prozeduraufrufe**): *Für einen Aufruf $p(a, \&b)$ der Prozedur (8.62) gilt mit (8.67)*

$$PR \to \mathrm{wp}(\text{„}p(a, \&b)\text{“}, S), \tag{8.68}$$

wenn die Prozedur keine Nebenwirkungen auf andere Variable als b hat.

Wir begnügen uns also mit der stärkeren Vorbedingung *PR*, um den Beweis von $\{P\}\ A\ \{Q\}$ für den Prozedurrumpf nur einmal führen zu müssen. Die Folgerung $Q[w/y] \rightarrow S[w/b]$ beantwortet die Frage nach der Bedeutung von Q im Kontext des Aufrufs.

Beispiel 8.30: Wie in Beispiel 8.22 bedeutet eine Vorbedingung *wahr*: Die Eingabeparameter haben einen beliebigen Wert. Beim Beweis eines Prozedurrumpfes folgern wir aus *wahr* Aussagen $x = x_0$ mit beliebigen Eingabewerten x_0 für alle Eingabeparameter x. In der Prozedur

```
erhöhe(i: INT; & k: INT)
 -- Vor:  wahr
 -- Nach:  k = i + 1
 is k := i+1 end
```

gilt daher die triviale Vorbedingung $i = i_0$ mit beliebigem i_0. In der Nachbedingung bezeichnet i den Anfangswert i_0. Für einen Aufruf erhöhe(a, &x) mit Nachbedingung S: $x > 4$ liefert unser Satz

$$\begin{aligned} PR\ :\ & (i = i_0)[a/i] \wedge \forall w\colon ((k = i_0 + 1)[w/k] \rightarrow (x > 4)[w/x]) \\ & = (a = i_0) \wedge \forall w\colon ((w = i_0 + 1) \rightarrow (w > 4)). \end{aligned}$$

Der zweite Term dieser Konjunktion ist nur für $i_0 > 3$ wahr. Zusammen mit dem ersten Term ergibt sich die Vorbedingung $a > 3$. ♦

In vielen Fällen sind Q und S mit Ausnahme der Bezeichner y bzw. b für das Ergebnis identisch. Da wir nur die Folgerung $Q[w/y] \rightarrow S[w/b]$ benötigen, in der die unterschiedlichen Bezeichner einheitlich durch w ersetzt sind, ist dann der zweite Term von *PR* eine Tautologie.

Beispiel 8.31: Die Prozedur

```
tausche'(i,j: INT; & a,b: INT)
 -- Vor:  wahr
 -- Nach:  a = j ∧ b = i
 is a:=j; b:=i end
```

weist die Anfangswerte i_0, j_0 vertauscht an die Ausgabeparameter a, b zu. Für einen Aufruf tausche'(u, v, &x, &y) mit Nachbedingung S: $(x = v) \wedge (y = u)$ erhalten wir aus (8.67)

$$\begin{aligned} PR\ :\ & (i = i_0) \wedge (j = j_0)[u/i, v/j] \wedge \\ & \quad \forall p, q\colon \big(\big((a = j_0) \wedge (b = i_0)\big)\,[p/a, q/b] \rightarrow \big((x = v) \wedge (y = u)\big)\,[p/x, q/y]\big) \\ & = (u = i_0) \wedge (v = j_0) \wedge \\ & \quad \forall p, q\colon \big(\big((p = j_0) \wedge (q = i_0)\big) \rightarrow \big((p = v) \wedge (q = u)\big)\big) \\ & = (u = i_0) \wedge (v = j_0), \end{aligned}$$

da die Implikation im zweiten Term gilt, wenn der erste Term wahr ist. ♦

Das Beispiel zeigt überdies, daß wir die gleichen Regeln auch für Prozeduren mit mehreren Eingabe- und Ergebnisparametern einsetzen können. Bei den Ergebnisargumenten haben wir jedoch zwei Annahmen gemacht, die wir explizit formulieren müssen:

1. Gibt es mehrere Ergebnisargumente, so müssen diese *verschiedene* Variable darstellen.
2. Die Nachbedingung S darf keine Aussagen über globale Variable enthalten, die in der Prozedur verändert werden und zugleich als Ergebnisargumente auftreten.

Wir bezeichnen diese Bedingungen mit disj(*arg*), wobei *arg* die Liste der Ergebnisargumente eines Prozeduraufrufs und der im Prozedurrumpf veränderten globalen Variablen ist.

Man sieht nun leicht, daß wir auch transiente Parameter zulassen können, indem wir sie als Paar von Eingabe- und Ergebnisparameter behandeln. Die Berechnung von *PR* in Beispiel 8.31 überträgt sich auf die Prozedur tausche von S. 29. Transiente Argumente müssen ebenfalls in die Liste *arg* der Ergebnisargumente aufgenommen werden.

Funktionsprozeduren lassen sich nach dem gleichen Schema verifizieren: Ein Aufruf $p(a, \&b)$ der Prozedur (8.62) führt zur gleichen Nachbedingung S wie die Zuweisung $b := f(a)$ mit der Funktionsvereinbarung

$$f(x\colon T)\colon T' \text{ is } A[\mathrm{res}/y] \text{ end} \tag{8.69}$$

in deren Rumpf wir den vormaligen Ergebnisparameter y durch die Ergebnisvariable res ersetzen.

Die Nachbedingung S eines Prozeduraufrufs $p(a, \&b)$ besteht häufig aus zwei Teilen $S = S' \wedge S''$. S' ist die eigentliche Nachbedingung des Prozeduraufrufs; S'' enthält keine Größen aus *arg* und ändert sich daher durch den Prozeduraufruf nicht. Daher kann S'' unverändert in die Vorbedingung übernommen werden, und wir erhalten abschließend

Korollar 8.9: *Sei arg die Liste der Ergebnisargumente, der transienten Argumente und der globalen Variablen, die durch einen Aufruf einer Prozedur (8.62) verändert werden. Es gelte* disj(*arg*)*, d. h. arg enthält keine Variable mehrfach.* $S = S' \wedge S''$ *sei eine Zusicherung, in der keine Variable aus arg in* S'' *vorkommt. Dann hat ein Aufruf* $p(a, \&b)$ *mit Nachbedingung S die Vorbedingung* $PR \wedge S''$ *mit*

$$PR\colon P[a/x] \wedge \forall w\colon (Q[w/y] \rightarrow S'[w/b]). \tag{8.70}$$

Aufgabe 8.32: Zeigen Sie $\{a = 4 \wedge b = 3\}$ updiv$(a, b, \&c)$ $\{c = 2\}$ mit

```
updiv (x,y: INT; & z: INT) is
  -- Vor:  x > 0 ∧ y > 0
  -- Nach: z − 1 < x div y ⩽ z
    z:= 0;
    while x > 0
    loop x := x - y; z := z + 1 end;
end;
```

Aufgabe 8.33: Zeigen Sie $\{a > 0\}$ sqrtint$(a, \&b)$ $\{b = \lfloor\sqrt{a}\rfloor\}$ mit

```
sqrtint(x: INT; & y: INT) is
  -- Vor:  x > 0
  -- Nach: y^2 ⩽ x < (y + 1)^2
  y := x;
  while y > x div y loop y := (y + x div y) div 2 end;
end;
```

Aufgabe 8.34 (Gries): Mit der Prozedur tausche aus Beispiel 8.10, vgl. auch Beispiel 8.31, gilt für einen Aufruf tausche(&& i, && $b[i]$)

$$\{i = i_0 \wedge (\forall j\colon b[j] = w_j)\}$$
$$\text{tausche}(\&\&\ i, \&\&\ b[i])$$
$$\{R\colon i = w_i \wedge b[i] = i_0 \wedge (\forall j\colon j \neq i_0 \rightarrow b[j] = w_j)\}$$

Aufgabe 8.35 (Gries): Die Prozedur

```
p(x: INT; && b: ARR[*](INT); && n: INT; & m: INT) is
-- Vor:  0 ⩽ n ∧ x = x0 ∧ b = b0
-- Nach: 0 ⩽ m < n ∧ b[m] = x0
  m := 0; b[n] := x;
  while x /= b[m]
  loop -- inv: 0 ⩽ m ⩽ n ∧ x ∉ b[0 : m − 1]
    m := m+1
  end;
  if m=n then
    n := n+1
  end
end; -- p
```

sucht x in der Reihung $b[0 : n-1]$. Wenn x nicht vorkommt, wird n erhöht und x in $b[0 : n-1]$ aufgenommen. Der Index m enthält zum Schluß die Position von x.

1. Spezifizieren die Vor- und Nachbedingung die Prozedur p vollständig? Oder gibt es zusätzliche Aussagen, die man über den Prozedurrumpf beweisen könnte?
2. Welche der folgenden Zusicherungen über einen Aufruf von p kann man mit Hilfe von Satz 8.8 beweisen (geeignete Vereinbarungen für f, c, s, j und Verallgemeinerung des Satzes auf mehrere, auch transiente Parameter vorausgesetzt)?:
   a. $\{wahr\}\ p(5, c, 0, j) \quad \{c[j] = 5\}$
   b. $\{0 \leqslant s\}\ p(f, c, s, j) \quad \{c[j] = f\}$
   c. $\{0 < s\}\ p(b[0], c, s, j)\ \{c[j] = c[0]\}$
   d. $\{0 < s\}\ p(5, c, s, m) \quad \{c[j] = 5\}$
3. Welche Beweise könnte man mit Korollar 8.9 führen?

8.2.7.1 Rekursives und iteratives Problemlösen

Die Funktionale until und while, also die Schleifen des funktionalen Programmierens, hatten wir in Abschnitt 5.5.1 als Spezialfälle rekursiver Funktionen eingeführt. Mit Hilfe von Zusicherungen können wir die Korrespondenz Schleife ↔ rekursive Funktion im zustandsorientierten Programmieren aufklären.

Beispiel 8.32 (**Binärsuche**): Gegeben sei eine Reihung a: ARR[n](INT), deren Elemente aufsteigend geordnet sind, $a[0] \leqslant a[1] \leqslant \cdots \leqslant a[n-1]$, sowie eine ganze Zahl x. Wir wollen wissen, ob x in a vorkommt. Mit sequentieller Suche ist der Aufwand O(n). Da aber a geordnet ist, können wir wie in Abschnitt 7.2.2 Teile-und-Herrsche mit einem Aufwand O(log n) anwenden. Die Übertragung der Lösung aus Abschnitt 7.2.2 lautet rekursiv:

```
suche_rek(x: INT; a: ARR[*](INT); u,o: INT): BOOL is
  -- Vor:   0 ⩽ u ∧ o ⩽ a.asize − 1
  -- Nach:  x ∈ a[u : o]?
  m: INT := (u+o) div 2;
  if u>o then res := false
  elsif x = a[m] then res := true
  elsif x<a[m] then res := suche_rek(x,a,u,m-1)
  else res := suche_rek(x,a,m+1,o) end -- Fall x>a[m]
end
```

Bei Verwendung von Listen hatten wir in der Rekursion die Listen verkürzt. Bei Verwendung von Reihungen führen wir stattdessen zusätzliche Indexparameter u, o ein, die uns den jeweils betrachteten Ausschnitt der Reihung zeigen. Der Aufruf „von außen“ lautet suche_rek($x, a, 0, a$.asize − 1). Will man diese zusätzlichen Parameter vermeiden, so muß man eine weitere (Haupt-)Prozedur

```
suche(x: INT; a: ARR[*](INT)): BOOL is res := suche_rek(x,a,0,a.asize-1) end
```

einführen, deren einziger Zweck darin besteht, die noch fehlenden Argumente für den rekursiven Aufruf von suche_rek zu ergänzen. suche_rek ist dann eine Hilfsprozedur, die dem Anwender nicht zur Verfügung stehen sollte.

Die vorstehende Aufteilung einer rekursiven Problemlösung in eine Haupt- und eine Hilfsprozedur kommt häufig vor. In Pascal oder Modula-2 vereinbart man die Hilfsprozedur lokal in der Hauptprozedur. Die Hilfsprozedur wird dadurch für den Anwender unsichtbar. In Sather, C++, Java oder C# benutzt man eine öffentliche und eine private Methode. Die öffentliche Methode besitzt die gewünschte Aufrufschnittstelle und delegiert die eigentliche Problemlösung an die private Methode.

Für eine Schleifenlösung bringen wir die rekursive Prozedur in die Form

```
suche_rek(x: INT; a: ARR[*](INT); u,o: INT): BOOL is
  m: INT := (u+o) div 2;
  if u<= o and x /= a[m]
  then
    if x<a[m] then o := m-1 else u := m+1 end;
    res := suche_rek(x,a,u,o)
  else res := u<=o -- aus u ⩽ o folgt x = a[m]!
  end
end
```

Aus dieser können wir, wie wir gleich zeigen werden, Programm 8.5 als Schleifenfassung herleiten. Es repräsentiert die Hauptprozedur mit eingebauter Hilfsprozedur. Aus der Hauptprozedur suche haben wir die Vorbesetzung lokaler Größen u, o übernommen. Dann haben wir eine Schleifenfassung von suche_rek explizit einkopiert. Dieses Einkopieren eines Prozedurrumpfs nennen wir im folgenden **offenen Einbau**.

Programm 8.5: Binärsuche

```
suche(x:INT; a:ARR[*](INT)): BOOL is
 -- Vor: wahr
 -- Nach: x ∈ a[0 : a.asize − 1]?
 u,o,m: INT;
 u := 0; o := a.asize-1; m := (u+o) div 2;
 while (u <= o) and (x /= a[m])
 loop
  if x < a[m] then o := m-1
  else u := m+1
  end;
  m := (u+o) div 2
 end;
 res := u <= o
end
```

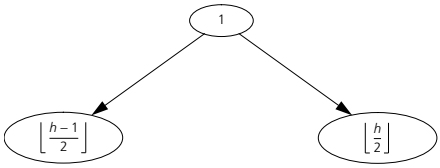

Abbildung 8.7: Schema eines binären Suchbaums

Für späteren Gebrauch analysieren wir, wie der Aufwand $T_{\text{suche}} = O(\log n)$ zustandekommt: Ein Durchlauf der Schleife zerlegt den Reihungsabschnitt $a[u : o]$ in die drei Teile $a[u : m-1]$, $a[m]$, $a[m+1 : o]$ des Umfangs $\lfloor\frac{h-1}{2}\rfloor$, 1, $\lfloor\frac{h}{2}\rfloor$ mit $h = o - u + 1$. Wir können dies durch den binären Suchbaum der Abb. 8.7, vgl. auch Abschnitt 6.3, darstellen. $\lfloor\frac{h-1}{2}\rfloor$ bzw. $\lfloor\frac{h}{2}\rfloor$ ist die Anzahl der Ecken im linken bzw. rechten Unterbaum. Setzen wir das Schema wie in Abb. 8.8 fort, so entspricht die Anzahl der Schleifendurchläufe der Weglänge+1 im Baum bis zum gefundenen Element. Ist x nicht im Baum enthalten, so ist für $2^{k-1} \leqslant n < 2^k - 1$ die Anzahl der Schleifendurchläufe $k-1$ oder k, da der Suchbaum die maximale Höhe (= maximale Weglänge) $k-1$ hat. Im Erfolgsfall kann die Anzahl der

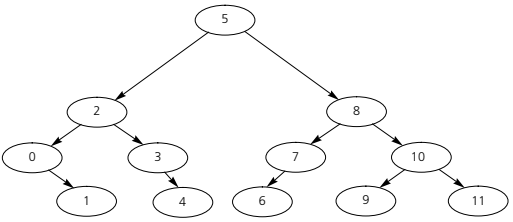

Abbildung 8.8: Binärer Suchbaum für $n = 11$

Durchläufe nicht größer sein, also gilt

$$T_{\text{suche}} \leqslant c_0 + c_1 \cdot (\lfloor \text{ld}\, n \rfloor + 1) = \text{O}(\log n). \tag{8.71}$$

Hier ist c_0 der Aufwand für den Prozeduranfang, c_1 der Aufwand für einen Schleifendurchlauf und $\lfloor \text{ld}\, n \rfloor + 1$ die maximale Anzahl von Wiederholungen. ♦

Obige Rekursion entspricht folgendem Schema, mit Bedingung B, Blöcken A, C.

$$p(x\colon T; \&y\colon T') \text{ is if } B \text{ then } A\,;\ x := f(x);\ p(x, \&y) \text{ else } C \text{ end end} \tag{8.72}$$

Dieses Schema heißt eine **rechtsrekursive Prozedur**. Es ist äquivalent zu

$$ps(x\colon T; \&y\colon T') \text{ is while } B \text{ loop } A;\ x := f(x) \text{ end};\ C \text{ end} \tag{8.73}$$

Die Zuweisung $x := f(x)$ spiegelt die Änderung der Argumente für den rekursiven Aufruf wider. Zum Beweis der Äquivalenz betrachten wir die Ausführungsreihenfolge der Ausdrücke, Zuweisungen und Blöcke in Abb. 8.9. Sie ist offenbar für (8.72) und (8.73) identisch. Die Vorbedingung P der rekursiven

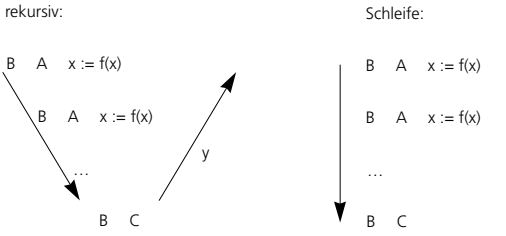

Abbildung 8.9: Rechtsrekursive und iterative Ausführung

Fassung ist in der rekursiven und der iterativen Fassung vor jeder Zeile erfüllt; in der iterativen Fassung ist P folglich eine Schleifeninvariante! Für die letzte Zeile erhalten wir die Zusicherung $\{P \wedge \neg B\}\ C\ \{Q\}$, wobei die Nachbedingung Q eine Aussage über das Ergebnis y enthält.

Die Aufgaben, eine Vorbedingung einer rekursiven Prozedur zu ermitteln und eine Schleifeninvariante zu finden, sind also in diesem Fall gleichwertig.

Beispiel 8.33 (**Hornerschema**): Gegeben seien die Koeffizienten $a_0, a_1, \ldots, a_n$ eines Polynoms $pol(x) = \sum_{i=0}^{n} a_i x^i$. Wir wollen den Polynomwert berechnen. Dazu sei $pol_k(x) = \sum_{i=k}^{n} a_i x^{i-k}$, $k = 0, \ldots, n$, also $pol(x) = pol_0(x)$. Es gilt

$$pol_k(x) = \begin{cases} a_n, & k = n, \\ x * pol_{k+1}(x) + a_k, & 0 \leqslant k < n. \end{cases} \tag{8.74}$$

Bilden die Koeffizienten eine Reihung a mit a.asize $= n + 1$, so erhalten wir:

```
horner(x: FLT; a: ARR[*](FLT); k: INT): FLT is
 -- Vor:  0 ⩽ k < a.asize
 -- Nach: res = pol[k](x)
 if k = a.asize-1 then res := a[k]
 else res := x*horner(x,a,k+1) + a[k]
 end
end
```

Der Aufruf horner(x, a, 0) berechnet $pol(x)$. Wie im vorigen Beispiel könnten wir eine Hauptprozedur einführen, um nach außen den zusätzlichen Parameter k zu verdecken. Mit einer Schleife erhalten wir Programm 8.6.

Programm 8.6: Hornerschema

```
horner(x: FLT; a: ARR[*](FLT)): FLT is
 -- Vor:  wahr
 -- Nach: res = pol[0](x)
 k: INT := a.asize-1;
 res := a[k];
 while k >= 1 loop
  k := k-1;
  res := x*res + a[k]
 end
end
```

Das rekursive Hornerschema entspricht mit den obigen Konventionen dem Schema

$$p(x\colon T; \&y\colon T') \text{ is if } B \text{ then } C \text{ else } x := f(x);\ p(x, \&y);\ A \text{ end end} \tag{8.75}$$

Allerdings setzen wir jetzt zusätzlich voraus, daß die Veränderungen $x := f(x)$ der Argumente zu einem Wert x_n führen, den wir vorhersagen können, und für den die Bedingung B nicht mehr erfüllt ist. Ferner soll die Berechnung von $f(x)$ keine Nebenwirkungen haben und umkehrbar sein. In unserem Beispiel entspricht $x := f(x)$ der Zuweisung $k := k + 1$ mit Endwert $k = a.\text{asize} - 1$ und Umkehrung $f^{-1}(k) = k - 1$.

Unter diesen Voraussetzungen heißt das Schema eine **linksrekursive Prozedur**. Es ist äquivalent zu

$$ps(x\colon T; \&y\colon T') \text{ is } C;\ \text{justasoften loop } x := f^{-1}(x);\ A \text{ end end} \qquad (8.76)$$

justasoften bedeutet, daß wir A ausführen, bis x wieder seinen Anfangswert erreicht hat, also für $x_i = f^{-1}(x_{i+1})$, $x_i = x_{n-1}, x_{n-2}, \ldots, x_0$. Die Abb. 8.10 zeigt, daß auch hier die Ausführungsreihenfolge identisch ist. Allerdings lassen

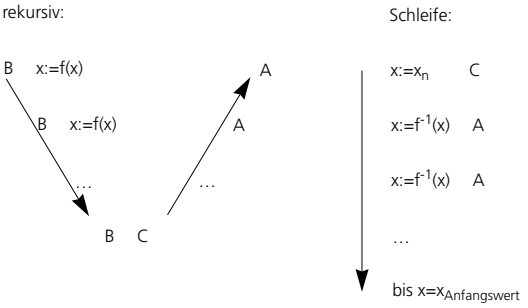

Abbildung 8.10: Linksrekursive und iterative Ausführung

wir die Anweisungen $x := f(x)$ aus und beginnen unmittelbar mit $x = x_n$. Man sieht, daß diesmal die Nachbedingung Q der rekursiven Prozedur der Schleifeninvariante der iterativen Fassung entspricht.

Für die Ausführung von C erhalten wir die Zusicherung $\{P \wedge \neg B\}\ C\ \{Q\}$.

Wenden wir das Schema auf eine Funktion statt auf eine eigentliche Prozedur an, so sieht man am Beispiel, daß in der Schleife statt des rekursiven Funktionsaufrufs die Ergebnisvariable res eingesetzt wird.

Die Transformation links- und rechtsrekursiver Prozeduren in Schleifen ist ein Spezialfall allgemeinerer Transformationen, die wir in Bd. III behandeln.

Aufgabe 8.36: Untersuchen Sie die Beispiele aus Kapitel 5, insbesondere aus Abschnitt 5.5.1: Welche sind rechts- bzw. linksrekursiv? Formulieren Sie imperative Fassungen dieser Funktionen in rekursiver und Schleifenform, und zeigen Sie ihre Äquivalenz. Bestimmen Sie die Vor- und Nachbedingungen beider Fassungen sowie die Invarianten der Schleifen.

Bei der Parameterübergabe wird in der rekursiven Fassung unserer Beispiele jedesmal die Reihung a der Länge n als Wertparameter übergeben und daher kopiert. Dies verursacht einen Aufwand von $\Theta(n)$. Einschließlich der Parameterübergabe ist der Aufwand für die Binärsuche daher nicht $O(\log n)$, sondern $O(n \log n)$. Für das Hornerschema beträgt er sogar $\Theta(n^2)$. In der Schleifenfassung erhalten wir für die beiden Beispiele $O(n + \log n)$ bzw. $\Theta(n)$; die Reihung wird jetzt nur noch einmal zu Beginn der Prozedur kopiert. Zwei Aufgabenlösungen

können also unterschiedlichen Ressourcenverbrauch und Zeitaufwand haben, auch wenn sie funktional gleichwertig sind.

Mit Hilfe von Referenzaufruf für die Übergabe der Reihung könnte man dieses Kopieren vermeiden und für die Binärsuche tatsächlich den Aufwand $O(\log n)$ erreichen. Dies ist die in Sprachen wie C, PASCAL, JAVA oder C# übliche Lösung des Problems. In Abschnitt 9.3.2 werden wir sehen, wie wir Referenzaufrufe in SATHER verwenden können.

8.2.8 Ausnahmebehandlung

Gegeben sei eine Zusicherung

$$\{P\}\ \textsf{begin}\ B\ \textsf{except}\ \textit{bez}\ \textsf{when}\ \text{AUSNAHMETYP}\ \textsf{then}\ A\ \textsf{end}\ \{Q\} \tag{8.77}$$

über eine Ausnahmeanweisung mit den Blöcken A, B entsprechend Abschnitt 8.1.6.5. Tritt die Ausnahme *bez* im Block B wirklich auf, so zerlegt sie die Ausführung des Blocks B in zwei Teile B', B'' von im allgemeinen unbekanntem Umfang: Einerseits könnte es mehrere Stellen geben, an denen die Ausnahme auftritt. Andererseits teilt uns die heutige Hardware oft nicht präzise mit, an welcher Stelle die Ausnahme auftrat, da zur gleichen Zeit mehrere Befehle ausgeführt wurden. Hardware oder Übersetzer könnten zudem die Reihenfolge der Operationen gegenüber dem Quellprogramm verändert haben, so daß selbst bei präziser Feststellung der Fehlerstelle keine klare Aufteilung des Quellprogramms in Teile B', B'' möglich ist.

Tritt die Ausnahme auf, so wird B' gefolgt von A ausgeführt. Damit erhalten wir ein Axiom für die Korrektheit der Zusicherung (8.77). Es muß gelten

$$\{P\}\quad B\quad \{Q\} \tag{8.78}$$

$$\{P\}\ B';\ A\,\{Q\}\ \text{für jede mögliche Aufteilung}\ B', B''\ \text{von}\ B \tag{8.79}$$

Will man nicht von speziellen Kenntnissen über die Hardware und den Übersetzer Gebrauch machen, sondern implementierungsunabhängig programmieren, so sind folgende Überlegungen nützlich:

1. Ausnahmeanweisungen können ohne Probleme nur eingesetzt werden, wenn, wie in dem Beispiel in Abschnitt 8.1.6.5, die Ausnahme zu einem speziellen Ergebnis führt, das in der Nachbedingung Q bereits vorgesehen ist.
2. Indem man von B Anfangs- und Endteile B_0, B_1 abspaltet, die den Fehler nicht verursachen und daher außerhalb der Ausnahmeanweisung stehen können, kann man oft B so verkleinern, daß die nach B' gültigen Zusicherungen genauer bestimmt werden können.
3. Läßt sich keine Aussage der Form (8.79) zeigen, so muß die Anweisung A die Ausnahme erneut auslösen, um in einer (dynamisch) umfassenden Ausnahmeanweisung eine ggf. schwächere Nachbedingung zu erfüllen.

Beispiel 8.34: Wird für eine Folge von Eingabedaten $e_1, e_2, \ldots$ unabhängig voneinander jeweils ein Ergebnis $f(e_i)$ berechnet und ausgegeben, so besagt Punkt 3: Wenn die Berechnung von $f(e_i)$ fehlschlägt, so sollte dies in einer Ausnahmeanweisung abgefangen werden, die zur Berechnung von $f(e_{i+1})$ überleitet. Die Nachbedingung „Fall e_i verarbeitet" läßt sich nämlich auf jeden Fall erfüllen. ♦

8.3 Anmerkungen und Verweise

Viele Erstveröffentlichungen über strukturiertes Programmieren und Weitergehendes findet man in (Gries, 1978). Dort findet sich insbesondere der Artikel (Hoare, 1969), der die Hoare-Logik einführt. Die Methodik der schwächsten Vorbedingungen wurde von (Dijkstra, 1976) eingeführt. Ein elementares Lehrbuch dazu ist (Dijkstra und Feijen, 1988); (Gries, 1981) gibt einen Gesamtüberblick mit vielen Beispielen. Die Verifikation von Prozeduren geht auf (Hoare und Wirth, 1973) zurück. Unsere Darstellung folgt (Gries, 1981).

Die Terminologie bezüglich Prozeduren ist relativ uneinheitlich, vgl. auch S. 6: In Maschinensprachen und in Fortran benutzt man das Wort **Unterprogramm** oder engl. *subroutine*. In der englischen Literatur über Eiffel oder Sather heißen Prozeduren **Routinen**. Unser Gebrauch des Worts Prozedur folgt der Terminologie von Sprachen wie Algol 60, Pascal und Modula-2.

http://www.springer.com/978-3-540-24403-5

Vorlesungen über Informatik
Band 2: Objektorientiertes Programmieren und Algorithmen
Goos, G.; Zimmermann, W.
2006, XIV, 375 S. 120 Abb., Softcover
ISBN: 978-3-540-24403-5