DIPLOMARBEIT

Symbolische Verifikation von Echtzeitprogrammen

ausgeführt am Institut für Automation

der Technischen Universität Wien

unter Anleitung von

Univ. Doz. Dr. Ulrich Schmid

und

Univ.Ass. Dipl.-Ing. Dr. Johann Blieberger

durch

Bernhard Scholz

Matr.Nr. 9127385

Bergsiedlung 5
3231 St. Margarethen

22. November 1996

Für meine Eltern

Zusammenfassung

Symbolische Verifikation von Echtzeitprogrammen

von Bernhard Scholz

Die in dieser Arbeit vorgestellte Thoerie verwendet symbolische Analysemethoden, um Echtzeitprogramme automatisch zu verifizieren.

Zeit ist in einem Programm eine implizite Größe. Sie ist von der Maschine, auf dem das Programm läuft, und vom Programm selbst abhängig.

Wir wandeln ein Programm in ein Zeitprogramm um. Die Zeit ist als Variable im Zeitprogramm enthalten, und für jede Anweisung wird sie um die Zeit der Ausführung der Anweisung erhöht. Am Ende der Berechnung steht nicht nur das Ergebnis, sondern auch die Zeit für die Auswertung des Ergebnisses zur Verfügung.

Die symbolische Analyse berechnet aus dem Zeitprogramm die Zeitfunktion. Ohne das Programm auszuführen, gibt die Zeitfunktion das zeitliche Verhalten des Programms an.

Mit geeigneten symbolischen Methoden kann die Spezifikation mit Hilfe der Zeitfunktion überprüft werden.

DANKSAGUNG

Besonderen Dank schulde ich Univ. Ass. Dipl.-Ing. Dr. Johann Blieberger, der mich zu diesem Thema inspiriert hat. Über ein Jahr lang setzte er sich intensiv mit meinen Ideen auseinander.

Ich stehe tief in der Schuld von Univ. Ass. Dipl.-Ing. Dr. Thomas Fahringer, der mir viele neue Anregungen zu diesem Thema gab.

Diese Diplomarbeit entstand im Rahmen des FWF-Projekts WOOP mit der Nummer P10188-MAT.

Weiters möchte ich mich bei allen WOOP-Projektmitgliedern bedanken, die mit Freundlichkeit meine unverständlichen Vorträge geduldig anhörten. Hier muß ich besonders Bernd Burgstaller erwähnen, mit dem ich viele Diskussionen über das Projekt führte.

Mein langjähriger Freund Reinhard Grafl gab mir zu Beginn der Arbeit viele Ratschläge.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

EINLEITUNG

Echtzeitprogramme unterscheiden sich von anderen Programmen: Sie müssen nicht nur richtige Ergebnisse liefern, sondern ihre Ergebnisse in der vorgesehenen Zeit berechnet haben. Es gibt aber zur Zeit keine Echtzeit-Werkzeuge, die vollautomatisch die zeitliche Korrektheit eines Programms überprüfen können. Jedoch ist gerade in "hard realtime systems" eine zeitliche Überprüfung unbedingt erforderlich. Ein Zeitfehler kann in diesen Systemen katastrophale Folgen haben.

Semiautomatische Methoden (vgl. [48, 59, 50]) wurden bereits vorgestellt. Sie haben den Nachteil, daß der Programmierer über Informationen, die im Programm enthalten sind, Aussagen treffen muß. Das Hinzufügen von bereits bestehender Information birgt Fehlerquellen in sich und ist für "hard realtime systems" nicht praktikabel.

Diese Arbeit stellt eine Theorie vor, die es ermöglicht, ein vollautomatisches Überprüfungsprogramm zu programmieren, das das Echtzeitverhalten von Programmen testet. In den folgenden Kapitel werden massiv NP-vollständige Formalismen erklärt. Die meisten Formalismen lassen sich nur mit komplizierten Algebrawerkzeugen(vgl. [63]) implementieren. Wir gehen aber davon aus, daß die statische Analyse eines Programmcodes weniger Zeit in Anspruch nimmt, als das Ausmessen und Überprüfen des gesamten Eingaberaums.

Mit Hilfe der symbolischen Analyse wird für ein Programm eine Zeitfunktion erstellt. Die Zeitfunktion berechnet für eine gegebene Eingabe die Ausführungszeit des Programms. Das zeitliche Verhalten kann daher, ohne das Programm auszuführen, bestimmt werden. Die symbolische Verifikation überprüft, ob für eine beliebige Eingabe die Zeitfunktion des Programms immer kleiner als die spezifizierte Zeit des Programms ist. Diese Art der Analyse erlaubt es uns, den Test auf Prozedurebene durchzuführen. Dadurch kann sehr viel Zeit für die Analyse eingespart werden. Der Aufwand ein komplettes Programm zu überprüfen, ist exponentiell höher als die einzelnen Prozeduren des Programms zu testen.

Wir verwenden eine imperative Minisprache τ-Simple. Sie wurde entwickelt, um die Formalismen in der symbolischen Analyse einfacher darzustellen. Die hier

vorgestellten Algorithmen können jederzeit auf kompliziertere Sprachen, wie Ada, Pascal oder C abgebildet werden.

Die Sprache τ-Simple besitzt nur zwei Datentypen: Arrays und beschränkte Skalare. Skalare können nur Zahlen aus einem endlichen Intervall der ganzen Zahlen sein. Arrays sind endlich in ihrer Länge und Dimension. Das prozedurale Konzept von τ-Simple ist sehr einfach gehalten. Prozeduren können sich gegenseitig aufrufen. Als Argumente von Prozeduren sind "Call by address" Variablen und "Call by value" Ausdrücke zugelassen. In Prozeduren können lokale Variablen deklariert werden, deren Werte zu Beginn undefiniert sind. Als Anweisungen sind die Null-Anweisung, die Zuweisung, die Hintereinanderausführung, die If-Anweisung und die While-Schleife erlaubt. In Ausdrücken können die Operatoren: $+, -, *$ und div verwendet werden. Eine Bedingung in einer Schleife oder If-Anweisung wird aus Relationen $=, \neq, <, >, \leq, \geq$, Junktoren (and, or) und dem Nicht-Operator zusammengesetzt. Ein-/Ausgabeanweisungen werden zunächst nicht behandelt. Die Eingabe erfolgt, indem die globalen Variablen eines Programms auf die gewünschten Werte gesetzt werden. Nach der Ausführung des Programm beinhalten die globalen Variablen die Ausgabe des Programms.

1.1 Überblick

Die Arbeit ist in zwei Kapitel aufgeteilt. Das erste Kapitel "Zeit und Echtzeit" beschreibt grundsätzliche Probleme der Zeit in Echtzeitprogrammen, und wie wir eine Zeitfunktion für ein gegebenes Programm erhalten. Weiters wird die Verifikation für Echtzeitprogramme skizziert.

Das zweite Kapitel beschäftigt sich ausschließlich mit "symbolischer Analyse". Es wurden einige neue Formalismen für die symbolische Analyse von Programmen entwickelt. Wir verwenden diese Formalismen, um die Zeitfunktion eines Programms zu finden. Diese Techniken wurden in den frühen 70ern für Programmverifikation entwickelt. Danach stoppte die Entwicklung und erst in den letzten Jahren wurde sie für unterschiedlichste Forschungsthemen wieder interessant (vgl. [31, 15, 14]).

Kapitel 2

ZEIT UND ECHTZEIT

2.1 Allgemein

Echtzeitprogramme müssen in der spezifizierten Zeit ihre Berechnungen abgeschlossen haben. Zeit ist aber eine implizite Eigenschaft eines Programms. Sie ist abhängig von der Maschine[1] und vom Programm selbst[2].

Um die Zeit für eine bestimmte Eingabe zu ermitteln, wird das Programm in ein Zeitprogramm transformiert. Das Zeitprogramm ist das ursprüngliche Programm, jedoch wird eine globale Zeitvariable hinzugefügt, die zu Beginn der Abarbeitung auf Null gesetzt wird. Für jedes Konstrukt im Programm, das Zeit für die Abarbeitung benötigt, wird die Zeitvariable erhöht. Nach der Auswertung eines Zeitprogramms steht uns nicht nur das Ergebnis zur Verfügung, sondern auch die Zeit, die für die Berechnung des Ergebnisses verwendet wurde.

Zunächst erscheint dieses Vorgehen sinnlos, da es in der heutigen Zeit exakte Uhren in fast jedem Echtzeit-Rechner gibt. Jedoch kann die Uhr nur während der Laufzeit (a posteriori) abgefragt werden. Das Zeitprogramm eines Programms hat aber die implizite Größe der Zeit explizit gemacht und durch geeignete symbolische Analysemethoden(vgl. [11],[51]) kann eine Zeitfunktion über der Eingabe gefunden werden. Diese Zeitfunktion erlaubt uns, für jede Eingabe, ohne das Programm auszuführen, die Zeit der Berechnung anzugeben.

Der Test, ob die Spezifikation erfüllt ist, kann mit dieser Methodik modularisiert werden. Die bisherigen Arbeiten (vgl. [58],[48]) verwenden eine konstante Zeitschranke und müßen daher das ganze Programm analysieren[3]. Unser Ansatz erlaubt es, auf Prozedurebene zu testen. Die Spezifikation ist eine Funktion über den Parametervariablen der Prozedur. Mit Hilfe der Zeitanalyse wird eine Zeitfunktion gefunden und es wird verifiziert, ob die Spezifikation immer größer gleich der Zeitfunktion des Programms ist.

---

[1] Auf einem schnellen Rechner läuft ein Programm schneller als auf einem langsamen.

[2] Eine Schleife, die 10 mal durchlaufen wird, braucht mehr Zeit als die selbe Schleife, die nur 5 mal durchlaufen wird.

[3] Auf Prozedurebene kann zwar mit konstanter Schranke getesten werden, hat aber den Nachteil, daß die Schranke zu hoch angesetzt werden muß, um korrekte Ergebnisse zu erhalten.

Für ein Programm bzw. eine Prozedur mit einer Zeitfunktion τ_P und einer Spezifikation τ_S muß gelten (X ist eine beliebige Eingabe):

$$\forall X : \tau_P(X) \leq \tau_S(X) \tag{2.1}$$

2.2 Zeit als Größe

Die Zeit ist in unserem Zeitmodell eine diskrete Größe. Wir gehen von einem Mikroprozessor aus, der mit einer konstanten Periode getaktet wird und die Dauer eines Maschinenbefehls kann in Taktzyklen exakt gemessen werden (Einheit $\cong$ Taktzyklus). Die Computerzeit ct wird mit einer perfekten Echtzeit rt synchronisiert.

$$ct = rt + \delta, |\delta| \leq \epsilon \tag{2.2}$$

Wobei ϵ die Genauigkeit der Computerzeit angibt (vgl. [39]). Da wir nur die internen Ereignisse — Programm startet bzw. stoppt — betrachten, haben wir nun das Problem, daß die Zeit der Spezifikation im System der perfekten Echtzeit und die Zeitfunktion des Programms in der Computerzeit modelliert sind.

Wir können beide Zeiten in einer a priori Analyse nur am Programmstart synchronisieren. Wir müssen daher für jede Anweisung einen zusätzlichen Aufschlag mit einrechnen, der die Schwankungen der Computerzeit berücksichtigt.

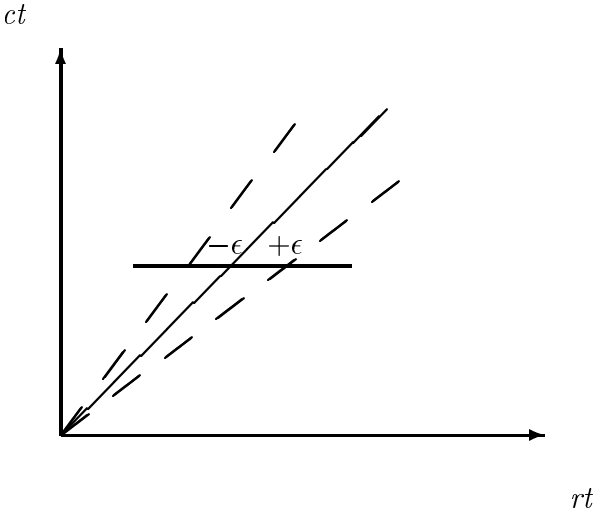


Abbildung 2.1: Perfekte Echtzeit vs. Computerzeit

Die Zeit ist in unserem Modell eine diskrete Größe und kann daher auf die natürlichen Zahlen abgebildet werden. Für Zeitintervalle verwenden wir folgende Notation:

- Ein Zeitintervall T besteht aus einer unteren und einer oberen Zeitschranke $[u, o]$. Es muß gelten: $u \leq o$.
- Eine Zeitdauer t ist im Zeitintervall T, wenn gilt:

$$t \in T \Longleftrightarrow u \leq t \leq o \tag{2.3}$$

- Die Addition zweier Zeitintervalle T_1, T_2 ist:

$$T_1 + T_2 = [u_1 + u_2, o_1 + o_2] \tag{2.4}$$

- Ein Zeitintervall T_1 ist im Zeitintervall T_2 enthalten, wenn gilt:

$$T_1 \subseteq T_2 \Longleftrightarrow u_1 \geq u_2 \wedge o_1 \leq o_2 \tag{2.5}$$

2.3 Zeitschema

Für ein Programm wird die Zeit mit Hilfe eines Zeitschemas $\mathcal{T}$ modelliert. Ohne die Laufzeit für eine gegebene Eingabe zu messen, ermittelt das Zeitschema die Zeit für die Ausführung des Programms. Wir erhalten eine symbolische Zeit, die keine physikalische Größe ist. Erst durch exaktes Messen kann das Zeitschema auf Korrektheit überprüft werden.

Eine vollständige Korrekheit ist erst dann gegeben, wenn für jedes Programm P in der Sprache und für alle Eingaben X, die symbolische Zeit gleich der gemessenen ist.

$$\forall P : \forall X : \mathcal{T}(P, X) \equiv \tau_P(X) \tag{2.6}$$

Die Funktion $\tau_p(X)$ ist eine Meßfunktion, die die exakte Laufzeit in Taktzyklen mißt. Da wir von einer deterministischen Maschine ausgehen, ist die Meßfunktion eine Funktion, die für ein Argumentenpaar P und X immer die gleiche Anzahl von Taktzyklen liefert. Die Zeit für die Ausführung kann Schwankungen unterworfen sein, da die Berechnung ein physikalischer Prozeß ist. Die Anzahl der Taktzyklen ist aber in einer beliebigen Wiederholung der Messung für ein gegebenes Paar P und X konstant und kann gezählt werden.

Die symbolische Zeit ist eine Größe im logischen Modell des Zeitschemas. Die Zeit der Meßfunktion ist eine reale physikalische Größe. Beide Zeiten können nur während der Ausführung des Programms bestimmmt werden (a posteriori).

Das Zeitschema macht aber die Zeit virtuell: Das Programm kann auf einer anderen Maschinen mit unterschiedlichem zeitlichen Verhalten ablaufen. Mit Hilfe des Zeitschemas kann trotzdem die Ausführungszeit der ursprünglichen Maschine ermittelt werden.

Es ist sehr aufwendig, ein Zeitschema zu erstellen, das die Forderung nach vollständiger Korrektheit erfüllt. Moderne Hardwarestrukturen verwenden Caches und Pipelines. Das zeitliche Verhalten von Caches und Pipelines ist deterministisch. Die Anzahl der Zustände ist aber immens groß, sodaß eine exakte Analyse fast nicht möglich ist. Zusätzlich werden in den heutigen Compilern aggressive Optimierungsstrategien eingesetzt, die den Programmfluß des erzeugten Maschinenprogramms stark verändern und eine Zeitanalyse erschweren.

Es ist daher einfacher, eine Abschätzung für die symbolische Zeit zu finden.

$$\forall P : \forall X : \tau_P(X) \in \mathcal{T}(P, X) \tag{2.7}$$

Ein Zeitintervall wird im Zeitschemata berechnet. Das Intervall gibt eine untere und obere Schranke für die symbolische Zeit an. Die gemessene Zeit muß daher im Intervall von $\mathcal{T}(P, X)$ liegen. Das allgemeinste Zeitintervall $[0, \infty]$ erfüllt die vollständige Korrektheit, ist aber nicht aussagekräftig genug. In den meisten Fällen können engere Zeitschranken gefunden werden.

Die Abschätzung sollte nicht global für die Bewertung eines sprachlichen Elements erfolgen, sondern für jede Ausprägung neu bewertet werden. Beispiel: Wenn ein Mikroprozessor einen speziellen Befehl für das Inkrementieren von Variablen hat und der Compiler eine Zuweisung der Form $v := v + 1$ optimiert, verwendet das Zeitschema die zeitliche Abschrankung der Zuweisung. Die Folge ist eine zu hohe Abschätzung für die Ausführungszeit der Zuweisung.

Eine globale Bewertung muß für alle möglichen Ausprägungen eines sprachlichen Elements die zeitliche Korrektheit erfüllen. Eine zu hohe Abschätzung ist die Folge. Engere Grenzen sind nur dann zu bekommen, wenn die Ausprägung eines Elements auf Maschinencodeebene zeitlich analysiert und dann in das Zeitschema eingesetzt wird. Wir benötigen entweder einen speziellen Compiler, der die Übersetzungsmuster zeitlich bewertet und diese Information dem Zeitschema übergibt oder es wird der ganze Maschinencode des Programms analysiert. Das Reengineering des Maschinencodes eines Programms ist allerdings sehr komplex und es wird in den seltensten Fällen eine direkte Zuordnung zu den sprachlichen Elementen im Hochspracheprogramm gefunden.

$E_1 \underline{op} E_2$	$\mathcal{T}_{\underline{op}} + \mathcal{T}(E_1) + \mathcal{T}(E_2)$
(E)	$\mathcal{T}_{(\,)} + \mathcal{T}(E)$
c	$\mathcal{T}_c$
v	$\mathcal{T}_v$
$v[E]$	$\mathcal{T}_{v[\,]} + \mathcal{T}(E)$

Tabelle 2.1: Ein Operator $\underline{op}$ ist ein Element aus der binären Operatorenmenge $\{+, -, *, \text{div}\}$.

$E_1 \underline{rel} E_2$	$\mathcal{T}_{\underline{rel}} + \mathcal{T}(E_1) + \mathcal{T}(E_2)$
C_1 and C_2	$\mathcal{T}_{\text{and}} + \mathcal{T}(C_1) + \mathcal{T}(C_2)$
C_1 or C_2	$\mathcal{T}_{\text{or}} + \mathcal{T}(C_1) + \mathcal{T}(C_2)$
not C	$\mathcal{T}_{\text{not}} + \mathcal{T}(C)$

Tabelle 2.2: Eine Relation $\underline{rel}$ ist ein Element aus der Relationenmenge $\{=, \neq, <, \leq, >, \geq\}$.

2.3.1 Eine naive Definition des Zeitschemas

In der folgenden Definiton verwenden wir eine globale Bewertung für das zeitliche Verhalten von sprachlichen Elementen, d.h., daß jede Zuweisung (in Abhängigkeit des Ausdrucks) den gleichen Zeitaufschlag erhält.

Ein Zeitschema ist eine Funktion über ein τ-Simple Programm S und eine Eingabe X. Das Ergebnis ist ein Intervall in den natürlichen Zahlen und bestimmt die obere und untere Grenze der symbolischen Zeit:

$$\mathcal{T} : S \times X \to \mathbb{N} \times \mathbb{N} \tag{2.8}$$

In den Tabellen 2.1, 2.2 und 2.3 sind die Zeitschemata für jedes sprachliche Konstrukt von τ-Simple angegeben. Prozeduren werden später behandelt. In der Definition der If- und While-Anweisung wird die Funktion val[†] verwendet. Sie gibt an, ob die If- bzw. Schleifen-Bedingung zu *true* oder *false* ausgewertet wird.

Beispiel. Das Beispiel in Abb. 2.2 addiert zur Variable x den Wert ein. Danach wird wieder die Variable x um zwei inkrementiert, falls x kleiner y ist. Andernfalls

[†]Im Unterkapitel 3.2 wird sie genau beschrieben.

$null$	$\mathcal{T}_{null}$
$var := E$	$\mathcal{T}_{var:=E} + \mathcal{T}(E)$
$vec[E_1] := E_2$	$\mathcal{T}_{vec[E_1]:=E_2} + \mathcal{T}(E_1) + \mathcal{T}(E_2)$
$S_1; S_2$	$\mathcal{T}_{;} + T(S_1) + \mathcal{T}(S_2)$
$if\ C\ then\ S_1\ else\ S_2$	$\mathcal{T}_{\text{if}} + \mathcal{T}(C) + \begin{cases} \mathcal{T}_{\text{then}} + \mathcal{T}(S_1), & \text{val}(C) = true \\ \mathcal{T}_{\text{else}} + \mathcal{T}(S_2), & \text{val}(C) = false \end{cases}$
$while\ C\ do\ S$	$\mathcal{T}_{while} + \mathcal{T}(C) +$ $\begin{cases} \mathcal{T}_{loop} + \mathcal{T}(while\ C\ do\ S), & \text{val}(C) = true \\ \mathcal{T}_{exit}, & \text{val}(C) = false \end{cases}$

Tabelle 2.3: Zeitschema für eine Anweisung.

```
decl x,y:[1..100];

.....

x:=x+1;
if x < y then
    x:=x+2
else
    y:=y+3
```

Abbildung 2.2: Beispielprogramm für das Zeitschema

wird die Variable y um drei erhöht.

Wir nehmen an, daß x den Wert 10 und y den Wert 20 besitzt. Wir berechnen die symbolische Zeit mit den Zeitkonstanten einer fiktiven Maschine. Die Konstanten sind in Tab. 2.4 aufgelistet.

Zunächst wird die Zeit für die Hintereinanderausführung der Zuweisung und der If-Anweisung ermittelt. Das Programm in Abb. 2.2 wird mit P abgekürzt.

$$\begin{aligned} \mathcal{T}(P) &= \mathcal{T}(S_1; S_2) \\ &= \mathcal{T}_{;} + \mathcal{T}(S_1) + \mathcal{T}(S_2) \\ &= [0, 5] + \mathcal{T}(S_1) + \mathcal{T}(S_2) \end{aligned}$$

Der Platzhalter S_1 ist die Zuweisung $x := x + 1$ und wird weiter mit Hilfe der

T_{op}	[5,10]
$T_{(\,)}$	[0,3]
T_v	[10,30]
T_c	[4,10]
$T_{v[\,]}$	[20,50]
T_{rel}	[4,10]
T_{and}	[4,10]
T_{or}	[4,10]
T_{not}	[4,8]

$T_{v:=E}$	[25,40]
$T_{v[E]:=E}$	[30,60]
$T_{;}$	[0,5]
T_{if}	[20,70]
T_{while}	[30,80]
T_{loop}	[3,8]
T_{exit}	[2,10]
T_{call}	[100,150]
T_{then}	[10,20]
T_{else}	[10,20]

Tabelle 2.4: Zeitkonstanten einer Maschine

Regeln des Zeitschemas aufgelöst. Für die Konstante 1 auf der rechten Seite der Zuweisung wird die Regel T_c angewendet, für die Variable x die Regel T_v und für die Addition die Regel T_{op}.

$$\begin{aligned}\mathcal{T}(P) &= [0,5] + \mathcal{T}_{var:=E} + \mathcal{T}(x+1) + \mathcal{T}(S_2) \\ &= [25,45] + \mathcal{T}_v + \mathcal{T}_{\text{op}} + \mathcal{T}_c + \mathcal{T}(S_2) \\ &= [44,95] + \mathcal{T}(S_2)\end{aligned}$$

Die zweite Anweisung ist eine If-Anweisung. In Abhängigkeit der Bedingung addiert das Zeitschema die Zeit des Then-Zweigs bzw. die Zeit des Else-Zweigs zu der Gesamtzeit auf. Die Variable hat vor der Ausführung der If-Anweisung den Wert 11 und die Variable y den Wert 20. Die Bedingung ist "x kleiner y", d.h., daß der Then-Zweig ausgeführt wird.

$$\begin{aligned}\mathcal{T}(P) &= [44,95] + \mathcal{T}_{\text{if}} + \mathcal{T}_{\text{then}} + \mathcal{T}(x<y) + \mathcal{T}(x := x+2) \\ &= [74,185] + \mathcal{T}_v + \mathcal{T}_{\text{rel}} + \mathcal{T}_v + \mathcal{T}(x := x+2) \\ &= [98,255] + \mathcal{T}(x := x+2)\end{aligned}$$

Die Zeit für die Zuweisung $x := x+2$ ist das Intervall $[44,90]$. Die fr die Ausfhrung bentigte Zeit liegt daher im Intervall:

$$\begin{aligned}\mathcal{T}(P) &= [98,255] + [44,90] \\ &= [132,345]\end{aligned}$$

Die hier ermittelte Zeit ist eine symbolische Zeit. Wenn die gemessene Zeit für die Ausführung des Programms mit der Eingabe $x = 10$ und $y = 20$ in diesem symbolischen Zeitintervall liegt, ist das Zeitschema für diese Eingabe korrekt.

2.4 Zeitprogramm

Das Zeitschema ist ein logisches Modell der symbolischen Zeit und kann nur während der Laufzeit die symbolische Zeit ermitteln. Das logische Modell wird in ein von τ-Simple berechenbares Modell umgewandelt.

Das Zeitprogramm (vgl. [51]) ist das ursprüngliche Programm, das mit einer Zeitvariable t erweitert wird. Vor der ersten Anweisung wird die Zeitvariable auf Null gesetzt. Für jede weitere Anweisung wird die Zeitvariable, um die Zeit erhöht, die diese Anweisung für die Ausführung benötigt. Die Zeiten der einzelnen Anweisungen werden aufbauend auf das Zeitschema ermittelt. Die symbolische Zeit einer Anweisung kann daher a priori bestimmt werden (vgl. [33, 46, 3, 44, 45]).

Tabelle 2.5 beinhaltet die Gammafunktion, die ein Programm in ein Zeitprogramm umwandelt. Für die Umwandlung muß bereits ein Zeitschema für eine Maschine vorhanden sein.

$$\Gamma : S \to S \tag{2.9}$$

Ausgehend von einem Zeitprogramm kann eine Zeitfunktion t_p berechnet werden. Mit geeigneten Analysemethoden (siehe "Symbolische Analyse") kann eine Funktion für die Zeitvariable gefunden werden. Diese Zeitfunktion berechnet die symbolische Zeit für eine gegebene Eingabe, ohne daß das Programm ausgeführt wird.

Beispiel. Das Beispiel in Abbildung 2.2 wird in ein Zeitprogramm umgewandelt (siehe 2.3). Die Zeiten der Anweisungen werden mit Hilfe der Zeitkonstanten aus Tabelle 2.4 ermittelt und in das Zeitprogramm eingesetzt.

2.5 Spezifikation und Verifikation

Das zeitliche Verhalten eines Programms muß überprüft werden. Wir gehen von einem prozeduralen Zeitmodell aus: Für jede Prozedur existiert eine zeitliche Spezifikation. Wenn die für die Ausführung benötigte Zeit immer kleiner der spezifizierten Zeit ist, ist das zeitliche Verhalten einer Prozedur korrekt. Wenn alle

$null$	$t = t + \mathcal{T}_{null}$; $null$
$var := E$	$t = t + \mathcal{T}_{var:=E} + \mathcal{T}(E)$; $var := E$
$vec[E_1] := E_2$	$t := t + \mathcal{T}_{vec[E_1]:=E_2} + \mathcal{T}(E_1) + \mathcal{T}(E_2)$; $vec[E_1] := E_2$
$S_1; S_2$	$t = t + \mathcal{T}_{;}$ $\Gamma(S_1); \Gamma(S_2)$
$if\ C\ then\ S_1\ else\ S_2$	$t := t + \mathcal{T}_{\text{if}} + \mathcal{T}_{(} C)$; $if\ C\ then$ $\quad t := t + \mathcal{T}_{\text{then}}; \Gamma(S_1)$ $else$ $\quad t := t + \mathcal{T}_{\text{else}}; \Gamma(S_2)$
$while\ C\ do\ S$	$t := t + \mathcal{T}_{while} + \mathcal{T}(C)$; $while\ C\ do$ $\quad t := t + \mathcal{T}_{\text{loop}}; \Gamma(S); t := t + \mathcal{T}_{\text{while}} + \mathcal{T}(C)$ $t := t + \mathcal{T}_{\text{exit}}$

Tabelle 2.5: Γ Transformationsfunktion

```
decl x,y:[1..100];
```

$t := [0, 0];$
.....

$t := t + \mathcal{T}_{:=} + \mathcal{T}_v + \mathcal{T}_{\text{op}} + T_c;$
`x:=x+1;`
$t := t + \mathcal{T}_{\text{if}} + \mathcal{T}_v + \mathcal{T}_{\text{rel}} + \mathcal{T}_v;$
`if x < y then`
  $t := t + \mathcal{T}_{\text{then}};$
  $t := t + \mathcal{T}_{:=} + \mathcal{T}_v + \mathcal{T}_{\text{op}} + T_c;$
  `x:=x+1`
`else`
  $t := t + \mathcal{T}_{\text{else}};$
  $t := t + \mathcal{T}_{:=} + \mathcal{T}_v + \mathcal{T}_{\text{op}} + T_c;$
  `y:=y+1`

Abbildung 2.3: Beispiel eines Zeitprogramms

Prozeduren und das Hauptprogramm in einem Programmsystem zeitlich korrekt sind, ist das Programmsystem ein Echtzeitprogramm.

Das Testen der Echtzeitfähigkeit auf Prozedurebene beschleunigt die Analyse. Wir müssen davon ausgehen, daß jede vollautomatische Echtzeitanalyse ein NP-vollständigs Problem ist und daß die Analyse sehr viel Zeit in Anspruch nimmt. Würde ein Programm zur Gänze getestet werden, würde die Analyse exponentiell länger dauern als die schrittweise Überprüfung jeder einzelnen Prozedur des Programms.

Wir haben in unserem Modell Ungenauigkeiten bis zur dritten Ordnung. Die Ungenauigkeit erster Ordnung entsteht durch die Schwankungen des Computerquarzes und wir müssen einen zusätzlichen Zeitaufschlag für die Zeitfunktion berücksichtigen. Die Ungenauigkeit zweiter Ordnung erhalten wir durch die naive Zeitmodellierung des Programms. Eine exakte Zeitfunktion ist nicht möglich, daher wird eine Abschätzung mit einer oberen und unteren Schranke für die Zeit verwendet. Die Ungenauigkeit dritter Ordnung entsteht durch die symbolische Analyse des Zeitprogramms. In vielen Fällen kann die Zeitfunktion automatisch nur geschätzt werden.

Diese Ungenauigkeiten führen zu groben Zeitschranken. Wir müssen aber diese Ungenauigkeiten in Kauf nehmen, um a priori die zeitliche Korrektheit zu prüfen.

2.5.1 Spezifikation und Verifikation

Die Spezifikation für eine Prozedur kann als unbedingte Funktion, als bedingte Funktion oder als Programm formuliert werden. Unbedingte Funktionen sind nur ein Spezialfall der bedingten Funktionen. Programme, die die spezifizierte Zeit für Prozeduren vorgeben, sind in der Verifikation schwierig zu behandeln. Die symbolische Analyse muß die Zeitvorgabe, die in Form eines Programms berechnet wird, in eine bedingte Funktion umwandeln.

Die Spezifikation einer Prozedur wird in den meisten Fällen vor dem Ausprogrammieren der Prozedur erstellt. Die Prozeduren können aber nur Bottom-Up auf zeitliche Korrektheit überprüft werden, da Ergebnisse einer aufgerufenen Prozedur das zeitliche Verhalten einer aufrufenden Prozedur verändern kann.

Beispiel. Das Beispiel in Abbildung 2.4 ist eine einfache Prozedur mit einem "Call by value" Parameter x. Die Prozedur durchläuft x Mal die While-Schleife. Die Spezifikation wird in den meisten Fällen vor dem Ausprogrammieren der Pro-

```
proc dummy(in x:[-100..100])
local y;

y:=1;
while y <= x
   y:=y+1
```

Abbildung 2.4: Programmbeispiel für die Spezifikation von Prozeduren

zedur erstellt. Wir können jetzt die Spezifikation auf drei unterschiedliche Arten angeben. Die erste Möglichkeit die Prozedur dummy zeitlich zu spezifizieren, ist eine unbedingte Funktion:

$$\tau_s(x) = [80 + 50|x|, 300 + 300|x|]$$

Die Spezifikation τ_s ist eine Funktion, die die maximal für die Eingabe x benötigte Anzahl von Taktzyklen für eine Maschine festlegt. Der Betrag von x wird deshalb genommen, da x negativ sein kann und die Zeitspezifikation immer positiv sein muß. Ist x negativ, liefert die Spezifikation zu hohe Schranken.

Wir können für die Spezifikation eine bedingte Funktion angeben. Sie würde exakter das zeitliche Verhalten der Prozedur dummy abschranken:

$$\tau_s(x) = \begin{cases} [80 + 50x, 300 + 300x], & x \geq 1 \\ [80, 300], & \text{sonst} \end{cases} \tag{2.10}$$

Die letzte Möglichkeit ist, daß die Spezifikation in Form einer Prozedur angegeben ist. Die Spezifikation berechnet die spezifizierte Zeit. Diese Möglichkeit ist sicherlich die mächtigste.

```
specproc dummy(in x:[-100..100],out t:time)
local y;

t:=[80,300]
y:=1;
while y <  x
   t:=t+[50,300];
   y:=y+1
```

Mit Hilfe der symbolischen Analyse wird die Spezifikationsprozedur in eine bedingte Funktion umgewandelt. Die Funktion würde exakt der letzten Spezifikationsfunktion entsprechen.

2.5.2 Verifikation

Die Verifikation ist nur dann korrekt, wenn das Zeitschema einer Maschine korrekt ist. Wir verwenden nicht die gemessene Zeit, sondern die symbolische Zeit. Wenn die gemessene Zeit einer Prozedur $\tau_p(X)$ und die Spezifikation $\tau_s(X)$ ist, muß gelten:

$$\forall X : \tau_p(X) \in \tau_S(X) \tag{2.11}$$

Da wir a priori Zeitaussagen nicht messen können, müssen wir anstatt der Meßfunktion τ_p, die Zeitfunktion t_p nehmen.

$$\forall X : t_p(X) \subseteq \tau_S(X), \quad \text{wenn } \tau_p(X) \in t_p(X) \tag{2.12}$$

Wir nehmen an, daß die Zeitfunktion t_p und die Spezifikation τ_S bedingte Funktionen sind.

$$t_p(X) = \begin{cases} t_1, & tc_1 \\ \vdots & \\ t_n, & tc_n \end{cases}$$

$$t_s(X) = \begin{cases} s_1, & sc_1 \\ \vdots & \\ s_m, & sc_m \end{cases}$$

Wenn wir in die Ungleichung die bedingten Funktionen einsetzen und erhalten wir folgende Formel:

$$\forall X : \begin{cases} t_1, & tc_1 \\ \vdots & \\ t_n, & tc_n \end{cases} \subseteq \begin{cases} s_1, & sc_1 \\ \vdots & \\ s_m, & sc_m \end{cases}, \quad \text{wenn } \tau_p(X) \in t_p(X). \tag{2.13}$$

Die Ungleichung können wir so umformen, daß für jedes t_i und s_j einzeln der Vergleich durchgeführt wird.

$$\forall X : \forall i : \forall j : s_i \subseteq t_j, \quad \text{wenn } tc_i \wedge sc_j$$

Die bedingte Ungleichung kann in ein logisches Prädikat umgewandelt werden.

$$\forall X : \forall i : \forall j : (s_i \subseteq t_j) \leftarrow (tc_i \wedge sc_j)$$

Wenn die Bedingung der Ungleichung hält, muß auf jedenfall die Ungleichung wahr sein.

Mit Hilfe von geeigneten symbolischen Methoden (vgl. [49, 21, 22]) kann der Test durchgeführt werden.

Beispiel. Für das Beispiel 2.4 wird das Zeitprogramm mit den konstanten aus Tabelle 2.4 erstellt. Mit Hilfe der symbolischen Analyse erhalten wir eine Zeitfunktion:

$$t_p(x) = \begin{cases} [85 + 101x, 190 + 233x], & x \geq 1 \\ [85, 190], & \text{sonst} \end{cases}$$

Die Spezifikation ist mit der bedingten Funktion

$$\tau_s(x) = \begin{cases} [80 + 50x, 300 + 300x], & x \geq 1 \\ [80, 300], & \text{sonst} \end{cases}$$

gegeben. Wir müssen jetzt prüfen, ob die Ungleichung

$$\forall x : t_p(x) \subseteq \tau_s(x)$$

für jede Eingabe hält. Wir setzten die bedingten Funktionen ein und erhalten

$$\forall x : \begin{cases} [85 + 101x, 190 + 233x], & x \geq 1 \\ [85, 190], & \text{sonst} \end{cases} \subseteq \begin{cases} [80 + 50x, 300 + 300x], & x \geq 1 \\ [80, 300], & \text{sonst} \end{cases}.$$

Danach formen wir die bedingte Ungleichung in eine unbedingte um,

$$\forall x : \begin{cases} [85 + 101x, 190 + 233x] \subseteq [80 + 50x, 300 + 300x], & x \geq 1 \\ [85, 190] \subseteq [80, 300], & \text{sonst} \end{cases}.$$

Der zweite Fall in der Fallunterscheidung ist eine Tautologie und braucht daher nicht mehr weiter untersucht werden. Der erste Fall ist etwas komplizierter. Beide Seiten der Ungleichungen sind Funktionen. Wenn wir für beide Seiten x explizit

machen, erhalten wir für die obere und untere Zeitschranke folgende Ungleichungen:

$$[85 + 101x, 190 + 233x] \subseteq [80 + 50x, 300 + 300x] \Longleftrightarrow$$
$$85 + 101x \geq 80 + 50x \wedge 190 + 233x \leq 300 + 300x \Longleftrightarrow$$
$$x \geq -\frac{5}{51} \wedge x \geq -\frac{110}{67}$$

Aus der Fallunterscheidung folgt, daß die Ungleichungen halten, wenn die Fallunterscheidung wahr ist. D.h., daß die zeitliche Spezifikation der Prozedur erfüllt ist.

Kapitel 3

SYMBOLISCHE ANALYSE

3.1 Einleitung

Symbolische Analyse ist ein Überbegriff von verschiedenen formalen Methoden, die eine statische Analyse von Programmen ermöglichen. In dieser Arbeit wurde sie zum Auffinden der Zeitfunktion eines Programms entwickelt. Die Zeitfunktion soll statisch (ohne das Programm auszuführen) ermittelt werden. Der Formalismus kann aber auch für andere Zwecke verwendet werden: Parallelisierung (vgl. [31, 61, 7]), Software-Reuse (vgl. [15, 14]), Programmverifikation (vgl. [16]), Programmvereinfachungen und Programmoptimierung sind Anwendungsgebiete für die symbolische Analyse von Programmen.

Ich habe versucht, die Begriffe, die zur Zeit in der Literatur verwendet werden, für meine Verwendung zu ordnen. Eine frühe Zusammenstellung von symbolischen Methoden findet man in [17].

Da diese Thematik noch Teil der Forschung ist, gibt die folgende Ausführung nur einen Einblick. Es gibt noch viele offene Fragen und die symbolische Analyse ist für Sprachen wie Ada (vgl. [35]), C(vgl. [37]) oder Pascal kaum anwendbar. Die Semantik dieser Sprachen ist zu komplex. Die symbolische Analyse ist aufgrund ihrer Verwandtheit zum Halteproblem (vgl. [34]) sehr komplex und damit in den meisten Fällen unentscheidbar. Wenn wir aber von der Analyse von Programmen auf Computer mit endlicher Zahlenarithmetik und endlichem Speicher ausgehen, können wir das Problem auf ein lösbares reduzieren. Es ist entscheidbar. Die Voraussetzung ist, daß keine Ein-/Ausgabe-Interaktionen in den zu analysierenden Programmen existieren. Die Eingabe muß vor dem Aufruf des Programms vollständig vorhanden sein. Am Ende der Ausführung liegt das Ergebnis bzw. die Ausgabe vor.

Aufbauend auf τ-Simple (eine imperative Minisprache) wird die symbolische Analyse vorgestellt.

3.2 Die Semantik von τ-Simple

Die Sprache τ-Simple ist eine imperative Minisprache. Für die symbolische Analyse wird eine Teilmenge vorgestellt und danach schrittweise erweitert. Zunächst enthält τ-Simple nur die Zuweisung, das If-Statement, die Hintereinanderausführung von zwei Statements und das Null-Statement. Die Variablen in τ-Simple sind diskrete Skalare.

Es ist für die symbolische Analyse notwendig, die Semantik der Sprache exakt zu beschreiben. Deshalb wird eine Evaluierungsfunktion vorgestellt, die in denotationeller Form (vgl. [28]) die Semantik definiert. Aufgrund der Einfachheit der Sprache ist sie leicht verständlich. Es gibt zunächst keine Prozeduren und kein Hauptprogramm. Die Eingabe erfolgt, indem die Variablen vor dem Programmstart mit den gewünschten Werten initialisiert werden. Wenn ein Fehler auftritt (eine Null-Division oder eine Zuweisung außerhalb des gültigen Zahlenbereichs), so ist die Evaluierungsfunktion für dieses Programm nicht definiert. Würde das Programm auf einen real exisitierenden Computer ablaufen, würde das einen Programmabbruch induzieren.

Ein Programm P ist theoretisch betrachtet eine Funktion $P(X)$, die die Eingabe X auf eine Ausgabe Y abbildet.

$$P(X) = Y \tag{3.1}$$

Die Eingabe X bestimmt die Variablenwerte zu Beginn des Programms. Nach der Ausführung enthalten die Variablen den Wert der Ausgabe Y. Der Vektor Y hat daher die gleiche Größe wie die Eingabe X. Es muß eine eindeutig Zuordnung der Variablen zu den Eingabekomponenten des Vektors X bzw. der Komponenten des Vektors Y gegeben sein.

Abbildung 3.1: Programmodell

Die Variablen können ihre Werte während der Ausführung des Programms ändern. Der momentane Wert einer Variable wird in der Variablenumgebung festgehalten. Nach der Ausführung enthält sie die Werte für die Ausgabe Y.

Die Evaluierungsfunktion wird mit Hilfe einer attributierten Grammatik (vgl. [1], [10], [43]) definiert. Die zweispaltigen Tabellen 3.1, 3.2, 3.3 enthalten in der

linken Spalte die rechten Seiten der Produktionen des jeweiligen Nonterminalsymbols und in der rechten die Evaluierung.

Die Grammatik von τ-Simple besteht zunächst aus drei verschiedenen Nonterminalsymbolen. S ist das Nonterminal für eine Anweisung und ist ein Platzhalter für ein If-Statement, für eine Zuweisung, für die Hintereinanderausführung oder für das Null-Statement. E wird als Nonterminal für einen arithmetischen Ausdruck verwendet. Das Symbol C steht für ein Konditional.

Je nach Typ des Nonterminals liefert die Evaluierungsfunktion entweder eine Variablenumgebung (bei Anweisungen), einen Booleschen Wert (bei Konditionalen) oder eine Zahl (bei arithmetischen Ausdrücken) als Ergebnis.

3.2.1 Variablenumgebung

Der momentane Wert einer Variable wird in einer Variablenumgebung env mitgeführt, die eine endliche Menge aus Paaren ist. Jedes Paar besteht aus dem Variablensymbol und einem Wert für die Variable. Jede Variable besitzt genau ein Paar in der Variablenumgebung.

$$\text{env} \in (\text{VS} \times \mathbb{Z})^n \tag{3.2}$$

VS ist die Menge der Variablensymbole und enthält alle Variablennamen des Programms. n ist die Anzahl der Variablen und ist für jedes Programm konstant. D.h, es ist nicht möglich, während der Laufzeit neue Variablen einzuführen. Die Wertemenge $\mathbb{Z}$ einer Variable ist ein Intervall aus den ganzen Zahlen. Das Intervall einer Variable legt die Funktion rg fest.

$$\begin{aligned} \text{rg} : \text{VS} &\to \mathbb{Z} \times \mathbb{Z} \\ \text{var} &\mapsto [u, o], u \leq o \end{aligned} \tag{3.3}$$

u und o sind die untere und obere Schranke der Variable var. Für die Definition der Hilfsfunktionen wird oft das Kürzel ENV verwendet. ENV ist die Menge aller Variablenumgebungen $(\text{VS} \times \mathbb{Z})^n$.

Zwei Hilfsfunktionen vget und vset werden benötigt. Sie ermöglichen den Zugriff auf die Variablenumgebung. vget liefert den Wert einer Variable in der Variablenumgebung. vset bildet eine Variablenumgebung auf eine neue ab. w ist

der neue Wert der Variable var.

$$\text{(3.4)}\quad \text{vget} : \text{ENV} \times \text{VS} \to \mathbb{Z}$$
$$\text{vget}\left(\{(v_1, w_1), \ldots, (v_{i-1}, w_{i-1}), (\text{var}, w_i), \ldots, (v_n, w_n)\}, \text{var}\right)$$
$$\mapsto w_i$$

$$\text{(3.5)}\quad \text{vset} : \text{ENV} \times \text{VS} \times \mathbb{Z} \to \text{ENV}$$
$$\text{vset}\left(\{(v_1, w_1), \ldots, (v_{i-1}, w_{i-1}), (\text{var}, w_i), \ldots, (v_n, w_n)\}, \text{var}, w\right)$$
$$\mapsto \{(v_1, w_1), \ldots, (v_{i-1}, w_{i-1}), (\text{var}, w), \ldots, (v_n, w_n)\}$$

Wenn auf eine Variable zugegriffen wird, die nicht in der Variablenumgebung vorhanden ist, so sind die Funktionen vset und vget nicht definiert.

3.2.2 Evaluierung eines Ausdrucks in τ-Simple

In τ-Simple berechnet die Funktion val_E den Wert eines arithmetischen Ausdrucks (siehe Tabelle 3.1).

$$\text{(3.6)}\qquad \text{val}_E : E \times \text{ENV} \to \mathbb{Z}$$

Aus Gründen der Lesbarkeit wird nicht zwischen Operatorsymbol und Operatoren unterschieden. *const* ist eine ganze Zahl und entspricht den Wert der Zahl in der Evaluierung. Der momentane Wert einer Variable wird mit Hilfe der Funktion vget aus der Variablenumgebung env entnommen. Die Divison $a \operatorname{div} b$ ist eine Integer-Division.

$$\text{(3.7)}\qquad a \operatorname{div} b = \operatorname{sgn}(a) \operatorname{sgn}(b) \left\lfloor \left| \frac{a}{b} \right| \right\rfloor$$

Wenn nur einer der beiden Operanden negativ ist, wird das Ergebnis nach oben abgeschnitten, andernfalls nach unten.

3.2.3 Evaluierung eines Konditionals in τ-Simple

Konditionale werden mit Hilfe der Funktion val_C ausgewertet. Ein Konditional kann einen Wert aus der Booleschen Menge $\mathbb{B}$ annehmen. $\mathbb{B}$ beteht aus den zwei Wahrheitswerten *true* und *false*. Tabelle 3.2 enthält Definitionen für die Evaluierung von Relationen, Junktoren (and, or) und der Negation (not).

$$\text{(3.8)}\qquad \text{val}_C : C \times \text{ENV} \to \mathbb{B}$$

Die Metavariable *rel* in der Tabelle 3.2 ist eines von den folgenden Relationssymbolen: $=, <, \leq, >, \geq, \neq$.

Syntax	*Semantik*
$E_0 + E_1$	$\mathrm{val}_E(E_0, \mathrm{env}) + \mathrm{val}_E(E_1, \mathrm{env})$
$E_0 - E_1$	$\mathrm{val}_E(E_0, \mathrm{env}) - \mathrm{val}_E(E_1, \mathrm{env})$
$E_0 * E_1$	$\mathrm{val}_E(E_0, \mathrm{env}) * \mathrm{val}_E(E_1, \mathrm{env})$
$E_0 \,\mathrm{div}\, E_1$	$\mathrm{val}_E(E_0, \mathrm{env}) \,\mathrm{div}\, \mathrm{val}_E(E_1, \mathrm{env})$, falls $\mathrm{val}_E(E_1, \mathrm{env}) \neq 0$
const	*const*
var	$\mathrm{vget}(\mathrm{env}, var)$

Tabelle 3.1: Evaluierung eines Ausdrucks in τ-Simple

Syntax	*Semantik*	
$C_0 \,\mathrm{rel}\, C_1$	*true*,	$\mathrm{val}_E(C_0, \mathrm{env}) rel\, \mathrm{val}_E(C_1, \mathrm{env})$
	false,	*sonst*
$C_0 \,\mathrm{and}\, C_1$	*true*,	$\mathrm{val}_C(C_0, \mathrm{env}) = true \wedge \mathrm{val}_C(C_1, \mathrm{env}) = true$
	false,	*sonst*
$C_0 \,\mathrm{or}\, C_1$	*true*,	$\mathrm{val}_C(C_0, \mathrm{env}) = true \vee \mathrm{val}_C(C_1, \mathrm{env}) = true$
	false,	*sonst*
$\mathrm{not}\, C$	*true*,	$\mathrm{val}_C(C, \mathrm{env}) = false$
	false,	*sonst*

Tabelle 3.2: Evaluierung eines Konditionals in τ-Simple

3.2.4 Evaluierung einer Anweisung

Anweisungen werden anhand Tabelle 3.3 ausgewertet.

$$\mathrm{val}_S : S \times \mathrm{ENV} \to \mathrm{ENV} \tag{3.9}$$

Das Null-Statement ist die einfachste Anweisung. Die übergebene Variablenumgebung wird unverändert zurückgegeben. Das Statement hat keine Auswirkung auf den Kontrollfluß und verändert die Daten nicht. Die Hintereinanderausführung von zwei Statements evaluiert zunächst die erste Anweisung S_1. Die daraus resultierende Variablenumgebung wird als Argument für das zweite Statement verwendet. Die If-Anweisung berechnet in Abhängigkeit des Konditionals entweder den Then- oder den Else-Zweig. Das Konditional wird zuerst evaluiert. Wenn die

Syntax	*Semantik*
null	env
$S_1; S_2$	$\mathrm{val}_S(S_2, \mathrm{val}_S(S_1, \mathrm{env}))$
if C *then* S_1 *else* S_2	$\begin{cases} \mathrm{val}_S(S_1, \mathrm{env}), & \mathrm{val}_C(C, \mathrm{env}) = \mathit{true} \\ \mathrm{val}_S(S_2, \mathrm{env}), & \mathit{sonst} \end{cases}$
$\mathit{var} := E$	$\mathrm{vset}(\mathrm{env}, \mathrm{var}, \mathrm{val}_E(E, \mathrm{env}))$, falls $\mathrm{val}_E(E, \mathrm{env}) \in \mathrm{rg}(\mathrm{var}, \mathrm{env})$

Tabelle 3.3: Evaluierung einer Anweisung in τ-Simple

Evaluierung *true* ergibt, wird der Then-Zweig genommen, andernfalls der Else-Zweig. Die Zuweisung verwendet die Funktion vset um den Wert der Variable var in der Variablenumgebung zu verändern. Die Evaluierungsfunktion ist aber nur dann definiert, wenn das Ergebnis des Ausdrucks innerhalb der Schranken der rg-Funktion liegt.

3.2.5 Funktionale Definition

Ein Programm $P(X)$ ist durch das Tupel $\langle S, \mathrm{rg}, \theta \rangle$ eindeutig bestimmt. S ist ein Element aus den möglichen Anweisungen in τ-Simple, rg der Zahlenbereich sämtlicher Variablen und θ eine Abbildung der Eingabe auf eine Variablenumgebung.

θ ordnet jeder Variable in der Variablenumgebung eine Komponente des Vektors $\mathbb{Z}^n$ zu. Die Zuordnung ist beliebig[1].

$$\begin{aligned} \theta : & \mathbb{Z}^n \to ENV \\ & (x_1, \ldots, x_n) \mapsto \{(v_1, x_1), \ldots, (v_n, x_n)\} \end{aligned} \tag{3.10}$$

θ erzeugt eine Anfangsumgebung für die Evaluierungsfunktion.

Die Funktion θ^{-1} ist die Umkehrfunktion von θ. Sie wandelt eine Variablenumgebung in einen Vektor aus $\mathbb{Z}^n$ um.

$$\begin{aligned} \theta^{-1} : & ENV \to \mathbb{Z}^n \\ & \{(v_1, x_1), \ldots, (v_n, x_n)\} \mapsto (x_1, \ldots, x_n) \end{aligned} \tag{3.11}$$

[1]Für die Zuordnung kann die alphabetischen Ordnung der Variablennamen genommen werden. Die erste Variable im Alpahbet ist v_1, usw.

Die Zuordnung der Variablen zu den Komponenten des Vektors muß in θ und in θ^{-1} ident sein.

Ein Programm $P(X)$ kann somit als Funktion

$$P(X) = \theta^{-1}(\mathrm{val}_S(S, \theta(X))) \tag{3.12}$$

geschrieben werden. θ wandelt die Eingabe in eine Anfangsumgebung für eine Anweisung S um. Die Funktion val_S wertet das Programm aus und liefert als Resultat eine Variablenumgebung, die mit der Funktion θ^{-1} in einen Vektor aus $\mathbb{Z}^n$ transformiert wird und die Ausgabe Y des Programms ist.

Beispiel. Das Beispiel in Abbildung 3.2 zeigt ein einfaches Programm. Im Deklarationsteil des Programmes werden zwei Variablen definiert, deren Wertebereich die Zahlen zwischen 1 und 100 sind. Danach folgt ein If-Statement, dessen Then-Zweig nur dann ausgeführt wird, wenn x kleiner als y ist. Im Then-Zweig weist das Programm der Variable y den Wert 10 zu. Wenn x größer gleich y ist, erhält x den Wert von y. Nach der If-Anweisung folgt eine null-Anweisung.

```
decl x,y:[1..100];

if x < y then
     y := 10
else
     x := y;
null
```

Abbildung 3.2: Beispielprogramm in τ-Simple

Der Deklarationsteil definiert die Zahlenbereiche der zwei Variablen x und y.

$$\mathrm{rg}(x) = [1, 100], \quad \mathrm{rg}(y) = [1, 100]$$

θ wird hier nicht explizit angegeben. Der erste Parameter des Programm $P(\underline{x}, \underline{y})$ wird der Variable x, der zweite der Variable y zugeordnet. Damit die Parameter von den Variablen unterschieden werden können, sind sie unterstrichen.

Das Programm soll mit $P(10, 20)$ ausgeführt werden. D.h., die Anfangskonfiguration ist $x = 10, y = 20$. θ erzeugt daher folgende Umgebung:

$$\mathrm{env}_0 = \{(x, 10), (y, 20)\}\,.$$

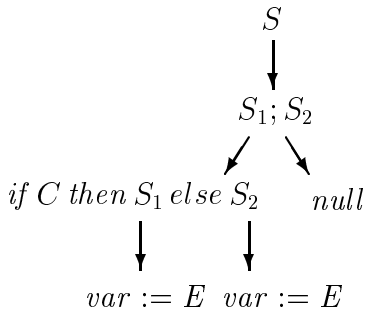


Abbildung 3.3: Syntaxbaum

Mit env_0 wird das Programm evaluiert. Das erste Konstrukt, das im Programm ausgewertet wird, ist die Hintereinanderausführung. Hier ist für die Abarbeitung die syntaktische Topologie (vgl. [34]) entscheidend — nicht die lexikographische Reihenfolge.

$$\begin{aligned} &\mathrm{val}_S(\mathit{if}\ x < y\ \mathit{then}\ y := 10\ \mathit{else}\ x := y; \mathit{null}, \mathrm{env}_0) = \\ &\mathrm{val}_S(\mathit{null}, \mathrm{val}_S(\mathit{if}\ x < y\ \mathit{then}\ y := 10\ \mathit{else}\ x := y, \mathrm{env}_0)) \end{aligned} \tag{3.13}$$

Jetzt wird die innerste val-Funktion weiter entwickelt. Die Evaluierungsfunktion wertet das Konditional $x < y$ aus. Dieses ist zusammengesetzt und besteht aus zwei Ausdrücken x und y, die mit der Evaluierungsfunktion für Ausdrücke weiterbehandelt werden. Mit Hilfe der Funktion vget werden die Werte der beiden Variablen gelesen. (In env_0 ist der Wert von x gleich 10 und der Wert von y gleich 20.)

$$\begin{aligned} \mathrm{val}_C(x < y, \mathit{env}_0) &= \mathrm{val}_E(x, \mathit{env}_0) < \mathrm{val}_E(y, \mathit{env}_0) \\ &= \mathrm{vget}(\mathit{env}_0, x) < \mathrm{vget}(\mathit{env}_0, y) \\ &= 10 < 20 \\ &= \mathit{true} \end{aligned}$$

Die Auswertung des Konditionals hat *true* ergeben. D.h., daß die Evaluierungsfunktion mit dem Then-Zweig weiterarbeitet. Der Then-Zweig wird noch immer mit env_0 evaluiert.

$$\begin{aligned} \mathrm{env}_1 &= \mathrm{val}_S(y := 10, \mathrm{env}_0) \\ &= \mathrm{vset}(\mathrm{env}_0, y, 10) \\ &= \{(x, 10), (y, 10)\} \end{aligned} \tag{3.14}$$

Die Funktion vset kann angewendet werden, weil der Variable y ein Wert im gültigen Bereich zugewiesen bekommt[2]. Entwickeln wir Formel 3.13 weiter, so wird die If-Anweisung durch env_1 ersetzt. (Die Berechnung des Then-Zweiges erfolgte in 3.14.)

$$\begin{aligned}\text{env}_2 &= \text{val}_S(\mathit{null}, \text{val}_S(\mathit{if}\ x < y\ \mathit{then}\ y := 10\ \mathit{else}\ x := y, \text{env}_0)) \\ &= \text{val}_S(\mathit{null}, \text{env}_1) \\ &= \text{env}_1\end{aligned}$$

Die Evaluierung der Null-Anweisung ist trivial. Die Variablenumgebung env_1 wird durchgereicht. D.h, das Ergebnis ist env_1. Also sind die Wert der beiden Variablen am Ende der Ausführung $x = 10, y = 10$. Das Programm P mit der Eingabe $(10, 20)$ hat als Ergebnis

$$\begin{aligned}P(\underline{x}, \underline{y}) &= P(10, 20) \\ &= (10, 10).\end{aligned}$$

3.3 USE/DEF-Mengen und der Kontrollflußgraph

3.3.1 USE/DEF-Mengen

Für die symbolische Analyse werden für einige Algorithmen die sogenannten USE- und DEF-Mengen (vgl. [1, 65]) benötigt. Sie geben für jedes Konstrukt in der Sprache an, welche Variablen möglicherweise verwendet bzw. definiert werden. Die Definition muß als zerstörende Operation gesehen werden: Der alte Wert einer Variable wird überschrieben. In τ-Simple existiert nur die Zuweisung, die genau eine Variable definiert. D.h., in der DEF-Menge der Zuweisung ist nur die Variable, der etwas zugewiesen wird, enthalten. Andere Anweisungen, die einen Seiteneffekt haben, existieren nicht. Jedoch in der prozeduralen Erweiterung von τ-Simple kann ein Prozedur-Aufruf mehreren Variablen neue Werte zuweisen.

Die Funktion für die Bestimmung der DEF-Menge ist in Abbildung 3.4 angegeben. Die DEF-Menge einer If-Anweisung wird berechnet, indem die DEF-Mengen beider Zweige vereinigt werden.

Die USE-Menge einer Anweisung, eines Konditionals bzw. Ausdrucks gibt die Menge der verwendeten Variablen an. In Abbildung 3.5 ist sie für alle sprachlichen Konstrukte dargestellt.

---

[2] $rg(y) = [1, 100]$

$$
\begin{array}{lcl}
DEF(null) & = & \{\} \\
DEF(S_1; S_2) & = & DEF(S_1) \cup DEF(S_2) \\
DEF(var := E) & = & \{var\} \\
DEF(if\ C\ then\ S_1\ else\ S_2) & = & DEF(S_1) \cup DEF(S_2)
\end{array}
$$

Abbildung 3.4: Definition von *DEF*

$$
\begin{array}{lll}
\forall \underline{OP} \in \{+, -, *, div, and, or\}: & USE(x\,\underline{OP}\,y) & = USE(x) \cup USE(y) \\
& USE(\text{not}\,x) & = USE(x) \\
& USE(var) & = \{var\}
\end{array}
$$

$$
\begin{array}{lcl}
USE(null) & = & \{\} \\
USE(S_1; S_2) & = & USE(S_1) \cup USE(S_2) \\
USE(var := E) & = & USE(E) \\
USE(if\ C\ then\ S_1\ else\ S_2) & = & USE(S_1) \cup USE(S_2)
\end{array}
$$

Abbildung 3.5: Definition von *USE*

3.3.2 KFG

Einige der folgenden Methoden benötigen für die Berechnung den Kontrollfluß-graph (KFG) bzw. den Begriff des Pfades. Ein KFG (vgl. [1]) ist ein gerichteter Graph $\langle N, E \rangle$. Jedes Statement n_i wird als Knoten im Graph dargestellt. N ist die Menge der Knoten.

$$N = \{n_1, n_2, \ldots, n_q\} \tag{3.15}$$

Der KFG besitzt genau einen Startknoten n_s und genau einen Endknoten n_f. Jede Kante im Graph ist ein geordnetes Paar (n_i, n_j) und bedeutet eine mögliche Hintereinanderausführung der beiden Statements n_j nach n_i. Die Menge der Kanten ist

$$E \subseteq N \times N. \tag{3.16}$$

Die Bedingung der Hintereinanderausführung wird durch das Sprungprädikat $\text{bp}(n_i, n_j)$ bestimmt. Für sequentiell hintereinandergereihte Anweisungen hat das Sprungprädikat den Booleschen Wert *true*. Für eine binäre Bedingung (If-Anweisung) muß das Sprungprädikat der einen Kante das Komplement der anderen sein.

Definition Ein KFG ist wohlgeformt, wenn für einen Knoten n_i, der genau zwei Knoten n_j, n_k vorangeht, gilt: $\mathrm{bp}(n_i, n_j) = \mathrm{not}\,\mathrm{bp}(n_i, n_k)$[†].

Wenn ein Knoten mehr als zwei Nachfolger hat, müssen alle Sprungprädikate disjunkt sein und es muß genau eines zu *true* evaluieren. In unserer Sprache τ-Simple tritt dieser Fall nicht auf, da die Sprache nur if-Anweisungen kennt[3].

Definition Die Menge der Nachfolger eines Knotens n_i ist die Menge der Knoten für die n_i unmittelbar vorangeht.

$$\mathrm{out}(n_i) = \{n_j | (n_i, n_j) \in E\} \tag{3.17}$$

Die Menge der Vorgänger wird analog definiert.

$$\mathrm{in}(n_i) = \{n_j | (n_j, n_i) \in E\} \tag{3.18}$$

Definition Ein Pfad ist eine endliche Folge von Knoten

$$\langle n_{i_0}, n_{i_1}, \ldots, n_{i_t} \rangle \tag{3.19}$$

und für jeden Knoten in der Folge existiert eine Kante zum nachfolgenden Knoten

$$\forall j, 0 \leq j < t : (n_{i_j}, n_{i_{j+1}}) \in E \tag{3.20}$$

Definition Ein Programmpfad π ist ein Pfad, der mit dem Startknoten beginnt und mit dem Endknoten endet.

$$\pi = \langle n_s, n_{i_0}, n_{i_1}, \ldots, n_{i_t}, n_f \rangle \tag{3.21}$$

Satz Jeder Knoten im KFG ist durch mindestens einen Programmpfad erreichbar.

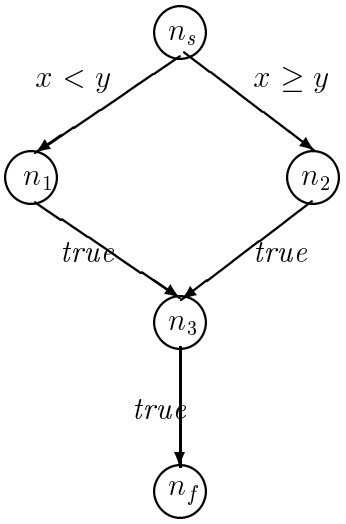
$$\forall : n_i \in E : \exists \pi : n_i \in \pi \tag{3.22}$$

Nicht erreichbare Knoten können sofort aus dem KFG genommen werden. Sie können durch keinen Pfad ausgeführt werden.

Beispiel 3.2 ist in der Abbildung 3.6 dargestellt. Jeder Knoten ist durch eine Marke $\{n_s, n_1, n_2, n_3, n_f\}$ eindeutig bestimmt, und jede Kante hat ein Sprungprädikat.

[†]Es darf keinen dritten Knoten n_l geben, für den gilt: $(n_i, n_l) \in E \wedge n_j \neq n_l \wedge n_k \neq n_l$.

[3]Durch das Einführen einer Case-Anweisung könnten Mehrfachverzweigungen im KFG enstehen.



Knoten	Anweisung
n_1	$y := 10$
n_2	$x := y$
n_3	*null*

Abbildung 3.6: Beispiel eines KFG

3.4 Symbolische Evaluierung

Die symbolische Evaluierung (vgl. [13, 17]) ist eine Datenflußanalyse. Sie untersucht statisch das dynamische Verhalten eines Programms. Ohne das Programm mit einer gegebenen Eingabe auszuführen, werden in einem beliebigen Punkt im Programm für die Variablen symbolische Werte berechnet. Ein symbolischer Wert ist Element einer Funktionenalgebra $\mathbb{F}$ (vgl. [42]).

Die Algebra besteht aus Grundfunktionen und Operatoren. Grundfunktionen können Konstante in $\mathbb{Z}$ bzw. Eingabeparameter sein. Die Operatoren entsprechen der Semantik in der Sprache τ-Simple und werden in zwei Klassen eingeteilt.

1. $\mathbb{F}_E$ ist die Klasse der Funktionen der Form $\mathbb{Z}^k \to \mathbb{Z}$. Das Resultat ist ein Element aus $\mathbb{Z}$. Bsp.: Addition von zwei Zahlen, ...

2. $\mathbb{F}_C$ ist die Klasse der Funktionen der Form $\mathbb{B}^k \to \mathbb{B}$ und $\mathbb{Z}^k \to \mathbb{B}$. Bsp.: Konjunktion und die Kleiner-Relation.

Die symbolische Evaluierung ist eine Übersetzung eines imperativen Programms in eine funktionale Sprache bzw. Funktionenalgebra (vgl. [38]).

$$\begin{array}{ccc} (P(X), K) & \longrightarrow & (\mathrm{sval}(P(X)), K) \\ \downarrow & & \downarrow \\ P(K) & = & \mathrm{sval}(P(K)) \end{array} \tag{3.23}$$

Ein Programm $P(X)$ mit den formalen Parametern X wird symbolisch evaluiert. Das Resultat der symbolischen Evaluierung $\text{sval}(P(X))$ ist ein Element der Funktionenalgebra (bzw. funktionalen Sprache). Wenn das Programm mit der Eingabe K ausgeführt wird, ist das Resultat das gleiche, wie wenn das symbolisch evaluierte Programm mit der gleichen Eingabe in der Funktionenalgebra ausgewertet wird.

Die Evaluierung in der Funktionenalgebra erfolgt für jede Ergebnisvariable getrennt. Für jedes y_i der Ausgabe findet die symbolische Evaluierung eine Funktion $f_{y_i}(X)$, die direkt das Ergebnis für diese eine Variable (unabhängig von den anderen) berechnet.

$$P(X) = (f_{y_1}(X), \ldots, f_{y_n}(X)) \tag{3.24}$$

Für die symbolische Evaluierung gibt es verschiedene Methoden, die für verschiedene Anwendungsgebiete mehr oder weniger geeignet sind. Sie unterscheiden sich in ihrer Effizienz und an der verwendeten Algebra. Das Grundprinzip bleibt aber das gleiche: Ein imperatives Programm wird in ein funktionales übersetzt. Für jedes Resultat des Programms existiert eine Funktion über den formalen Parametern.

Ähnlich wie bei der Semantikfunktion für τ-Simple entwickeln wir eine neue, symbolische Evaluierungsfunktion sval. Sie berechnet symbolisch die Werte der einzelnen Variablen. Die Wertemenge der Variablen ist eine Funktionenalgebra $\mathbb{F}_E$. Für die symbolische Evaluierung wird eine symbolische Variablenumgebung verwendet. Sie ist der Umgebung in der Standardsemantik ähnlich. Die Werte der Variablen sind aber symbolische Ausdrücke (Elemente aus $\mathbb{F}_E$).

$$\text{env} \in (\text{VS} \times \mathbb{F}_E)^n \tag{3.25}$$

Die symbolischen Ausdrücke können nach den Rechenregeln der Algebra $\mathbb{F}$ in Abbildung 3.7 vereinfacht werden.

Die symbolische Evaluierungsfunktion wird zunächst analog zur Semantikfunktion definiert. Der Unterschied liegt in der Verwendung einer anderen Wertemenge der Variablen.

Die If-Anweisung wird zunächst nicht behandelt, da die symbolische Evaluierungsfunktion, ohne die Eingabe zu kennen, nicht entscheiden kann, welcher der beiden Zweige ausgeführt wird. Beide Pfade sind relevant und werden für die Berechnung des Ergebnisses benötigt. Es gibt für die If-Anweisung mehrere Ansätze, die später vorgestellt werden.

Kommutivität :	$\forall i, j \in \mathbb{F}$
	$i + j = j + i$
	$i * j = j * i$
	$i \,\text{and}\, j = j \,\text{and}\, i$
	$i \,\text{or}\, j = j \,\text{or}\, i$
Assoziativität :	$\forall i, j, k \in \mathbb{F}$
	$i + (j + k) = (i + j) + k$
	$i * (j * k) = (i * j) * k$
	$i \,\text{and}(j \,\text{and}\, k) = (i \,\text{and}\, j) \,\text{and}\, k$
	$i \,\text{or}(j \,\text{or}\, k) = (i \,\text{or}\, j) \,\text{or}\, k$
Distributivität :	$\forall i, j, k \in \mathbb{F}$
	$i * (j + k) = (i * j) + (i * k)$
	$i \,\text{and}(j \,\text{or}\, k) = (i \,\text{and}\, j) \,\text{or}(i \,\text{and}\, k)$
	$i \,\text{or}(j \,\text{and}\, k) = (i \,\text{or}\, j) \,\text{and}(i \,\text{or}\, k)$
Identität :	$\forall i \in \mathbb{F}$
	$i + 0 = i$
	$i * 1 = i$
	$i \,\text{div}\, 1 = i$
	$i \,\text{and}\, \mathit{true} = i$
	$i \,\text{or}\, \mathit{false} = i$
Inverse :	$\forall i \in \mathbb{F}$
	$i - i = 0$
	$i \,\text{and not}\, i = \mathit{false}$
	$i \,\text{or not}\, i = \mathit{true}$
Eliminierung :	$\forall i, j, k \in \mathbb{F} \wedge i \neq 0$
	$i * j = i * k \Rightarrow j = k$
De Morgan :	$\forall i, j \in \mathbb{F}$
	$\text{not}(i \,\text{and}\, j) = (\text{not}\, i) \,\text{or}(\text{not}\, j)$
	$\text{not}(i \,\text{or}\, j) = (\text{not}\, i) \,\text{and}(\text{not}\, j)$
Inv. Relation :	$\forall i, j \in \mathbb{F}$
	$\text{not}(i < j) = (i \geq j)$
	$\text{not}(i > j) = (i \leq j)$
	$\text{not}(i = j) = (i \neq j)$
	$\text{not}(i \neq j) = (i = j)$
	$\text{not}(i \leq j) = (i > j)$
	$\text{not}(i \geq j) = (i < j)$

Abbildung 3.7: Rechengesetze der Funktionenalgebra $\mathbb{F}$

$$\text{sval}_S : S \times \text{ENV}_S \to \text{ENV}_S \tag{3.26}$$

$$\text{sval}_C : C \times \text{ENV}_S \to \mathbb{F}_C \tag{3.27}$$

$$\text{sval}_E : E \times \text{ENV}_S \to \mathbb{F}_E \tag{3.28}$$

ENV_S ist die Menge aller symbolischen Variablenumgebungen.

```
decl x,y:[1..100];

x = x + y;
y = x - y;
x = x - y
```

Abbildung 3.8: Algebraischer Dreieckstausch

Das Beispiel[4] in Abbildung 3.8 tauscht die Variablenwerte der beiden Variablen x und y[†]. Es ist nicht sofort ersichtlich, daß hier die Variablenwerte ausgetauscht werden. Wenn wir das Programm symbolisch evaluieren, wird der Sachverhalt klarer.

Mit Hilfe der θ-Funktion wird eine symbolische Variablenumgebung aus den Parametern erzeugt.

$$\text{env}_0 = \left\{(x, \underline{x}), (y, \underline{y})\right\}$$

Hier haben beide Variablen als Wert die formalen Parameter $\underline{x}$ und $\underline{y}$. Als nächstes wird das erste Statement im Beispiel Abbildung 3.8 ausgeführt.

$$\begin{aligned} &\text{sval}\,(x := x + y; y := x - y; x := x - y, \text{env}_0) = \\ &\text{sval}\,(y := x - y; x := x - y, \text{sval}\,(x := x + y, \text{env}_0)) \end{aligned} \tag{3.29}$$

Das Programm besteht aus zwei geschachtelten Hintereinanderausführungen. Die erste Anweisung wird separiert und symbolisch ausgeführt. Bei der Substitution der Variablenwerte muß darauf geachtet werden, daß die formalen Parameter nicht

[4]Dankend erwähne ich hier Boris Georgiew, der meine Aufmerksamkeit auf dieses Beispiel lenkte.

[†]Der Dreieckstausch wird normalerweise mit drei Variablen durchgeführt. Zuerst wird einer Hilfsvariable der Wert der ersten Variable zugewiesen. Danach bekommt die erste Variable den Wert der zweiten. Die zweite Variable übernimmt am Ende den Wert der Hilfsvariable.

für weitere Ersetzungen herangezogen werden. Es muß eine klare Unterscheidung zwischen den formalen Parameter $\underline{x}$ und der Variable x geben. Die Variable hat zu Beginn der Ausführung den Wert des Parameters, sie kann sich, während das Programm abgearbeitet wird, ändern.

$$\begin{aligned} \mathrm{env}_1 &= \mathrm{sval}\,(x := x + y, env_0) \\ &= \Big\{(x, \underline{x} + \underline{y}), (y, \underline{y})\Big\} \end{aligned}$$

Die neue symbolische Variablenumgebung env_1 wird in 3.29 eingesetzt, und die zweite Hintereinanderausführung wird aufgespalten.

$$\begin{aligned} &\mathrm{sval}\,(y := x - y; x := x - y, \mathrm{sval}\,(x := x + y, \mathrm{env}_0)) = \\ &\mathrm{sval}\,(y := x - y; x := x - y, \mathrm{env}_1) = \\ &\mathrm{sval}\,(x := x - y, \mathrm{sval}\,(y = x - y, \mathrm{env}_1)) \end{aligned} \tag{3.30}$$

Die zweite Zuweisung wird mit der Variablenumgebung env_1 symbolisch evaluiert.

$$\begin{aligned} \mathrm{env}_2 &= \mathrm{sval}\,(y := x - y, \mathrm{env}_1) \\ &= \Big\{(x, \underline{x} + \underline{y}), (y, \underline{x})\Big\} \end{aligned}$$

Jetzt kann die Variablenumgebung in die Formel 3.30 eingesetzt werden, und das letzte Statement wird ausgewertet.

$$\begin{aligned} \mathrm{env}_3 &= \mathrm{sval}\,(x := x - y, \mathrm{sval}\,(y := x - y, \mathrm{env}_1)) \\ &= \mathrm{sval}\,(x := x - y, env_2) \\ &= \Big\{(x, \underline{y}), (y, \underline{x})\Big\} \end{aligned}$$

Die Umgebung env_3 wird mit der θ^{-1} Funktion auf den Ausgabevektor abgebildet.

$$\begin{aligned} P(\underline{x}, \underline{y}) &= \theta^{-1}(env_3) \\ &= \theta^{-1}(\Big\{(x, \underline{y}), (y, \underline{x})\Big\}) \\ &= (\underline{y}, \underline{x}) \end{aligned}$$

3.4.1 If-Anweisungen in der Pfad-enumerierenden Methode

If-Anweisungen stellen in der symbolischen Auswertung ein Problem dar: Wir wissen nicht den Wert der Eingabe und können das Konditional in der If-Anweisung in den meisten Fällen nicht zu *true* oder *false* evaluieren.

In der Analyse einer If-Anweisung müssen daher beide Fälle symbolisch behandelt werden. Das If-Statement wird sowohl mit dem Then-Zweig als auch mit dem Else-Zweig symbolisch ausgewertet. Die Entscheidung, welcher Zweig genommen wird, bestimmt die Pfadbedingung. Sie wird aus den Konditionalen der If-Bedingungen abgeleitet. Nach der Auswertung der If-Anweisungen existieren zwei Variablenumgebungen. Beide sind gültig, da ohne die Eingabe zu kennen, nicht entschieden werden kann, welche verwendet wird. Jede nachfolgende Anweisung wird daher mit beiden Umgebungen evaluiert.

Für die Evaluierung (vgl. [17]) ist eine neue Datenstruktur notwendig, die die verschiedenen Variablenumgebungen mit ihren Pfadbedingungen aufnimmt: Der symbolische Kontext ctx ist eine endliche Menge aus Paaren. Ein Paar besteht aus einer Variablenumgebung und einer Pfadbedingung, die diese Variablenumgebung erzwingt.

$$\text{ctx} \in (\text{ENV}_S \times \mathbb{F}_C)^k \tag{3.31}$$

$\mathbb{F}_C$ ist die Menge aller Konditionale in der Funktionenalgebra, k ist die Kardinalität der Menge und gibt die Anzahl der Variablenumgebungen im Kontext an, CTX beschreibt die Menge $(\text{ENV}_S \times \mathbb{F}_C)^k$ und wird für die Definition der symbolischen Auswertungsfunktion pval benötigt.

Die Funktion pval führt eine Anweisung S mit einem Kontext ctx in einen neuen Kontext über.

$$\text{pval} : S \times \text{CTX} \to \text{CTX} \tag{3.32}$$

Auf den Kontext hat die Null-Anweisung keinen Seiteneffekt. Der übergebene Kontext wird durchgereicht.

$$\text{pval}(\mathit{null}, \text{ctx}) \mapsto \text{ctx} \tag{3.33}$$

Für die Zuweisung ist die Definition schwieriger. Der Kontext kann mehr als eine Variablenumgebung enthalten. Jede Variablenumgebung wird daher für die symbolische Auswertung der Zuweisung herangezogen. Die Funktion sval_S führt die Zuweisung mit einer Variablenumgebung aus[5].

$$\begin{aligned} &\text{pval}\,(\mathit{var} := E, \{(\text{env}_1, p_1), \ldots, (\text{env}_k, p_k)\}) \mapsto \\ &\{(\text{sval}_S(\mathit{var} := E, \text{env}_1), p_1), \ldots, (\text{sval}_S(\mathit{var} := E, \text{env}_k), p_k)\} \end{aligned} \tag{3.34}$$

[5]Eine vollständig neue Definition von pval ist daher nicht notwendig.

Die Hintereinanderausführung wird analog zur bisherigen Evaluierung definiert.

$$\text{pval}(S_1; S_2, \text{ctx}) \mapsto \text{pval}(S_2, \text{pval}(S_1, \text{ctx})) \tag{3.35}$$

Die If-Anweisung wertet beide Zweige aus. Jede Pfadbedingung einer Variablenumgebung wird mit der symbolisch ausgewerteten Bedingung der If-Anweisung erweitert. Für den Then-Zweig wird das evaluierte Konditional $\text{sval}(C, \text{env}_i)$ mit dem Und-Operator zu der Pfadbedingung p_i konjungiert. Die Pfadbedingungen des Else-Zweigs wird mit dem Negat des ausgewerteten Konditionals $\text{not}\,\text{sval}(C, \text{env}_i)$ erweitert. Das Resultat einer If-Anweisung sind die Variablenumgebungen des Then- und des Else-Zweigs.

(3.36)

$$\begin{aligned}
&\text{pval}(\mathit{if}\ C\ \mathit{then}\ S_1\ \mathit{else}\ S_2, \{(\text{env}_1, p_1), \ldots, (\text{env}_k, p_k)\}) \mapsto \\
&\text{pval}(S_1, \{(\text{env}_1, p_1 \,\text{and}\, \text{sval}_C(C, \text{env}_1)), \ldots, (\text{env}_k, p_k \,\text{and}\, \text{sval}_C(C, \text{env}_k))\}) \cup \\
&\text{pval}(S_2, \{(\text{env}_1, p_1 \,\text{and not}\, \text{sval}_C(C, \text{env}_1)), \ldots, (\text{env}_k, p_k \,\text{and not}\, \text{sval}_C(C, \text{env}_k))\})
\end{aligned}$$

Die Funktion θ und θ^{-1} müssen in der Pfad-enumerierenden Methode modifiziert werden. θ erzeugt für pval einen Kontext, der nur eine Variablenumgebung besitzt. Die Pfadbedingung der Umgebung ist *true*.

$$\theta(\underline{x_1}, \ldots, \underline{x_n}) \mapsto \{(\text{env}, \mathit{true})\} \tag{3.37}$$

env ist eine Variablenumgebung, die genau wie für das θ in sval bzw. val aufgebaut wird.

Für θ^{-1} gibt es mehrere mögliche Werte für eine Komponente des Ausgabevektors. Die dazugehörenden Pfadbedingungen bestimmen, welcher Wert genommen wird. Es gibt also für jede Komponente eine Fallunterscheidung.

$$\begin{aligned}
\theta^{-1}(\{ \quad & ((v_1, y_{11}) \quad , \ldots, \quad (v_n, y_{1n}), p_1), \\
& \vdots \qquad\qquad\quad \vdots \qquad\qquad\quad = \\
& ((v_{1n}, y_{k1}) \quad , \ldots, \quad (v_n, y_{kn}), p_k)\}) \mapsto \\
& \left(\begin{Bmatrix} y_{11}, & p_1 \\ \vdots & \\ y_{k1}, & p_k \end{Bmatrix} , \ldots , \begin{Bmatrix} y_{1n}, & p_1 \\ \vdots & \\ y_{kn}, & p_k \end{Bmatrix} \right)
\end{aligned} \tag{3.38}$$

```
decl x,y:[1..100]
if x  < y then
   x := y - x
else
   y := x - y;
x := x + y
```

Abbildung 3.9: Beispiel für die Pfad-enumerierende Methode

Beispiel. Das Beispiel in Abbildung 3.9 ist ein einfaches Programm, das der Variable x die Differenz von y und x zuweist, falls x kleiner y ist. Andernfalls wird die Differenz von x und y der Variable y zugewiesen. Die letzte Anweisung berechnet die Addition von x und y und weist das Ergebnis der Variable x zu.

Das Beispiel beinhaltet bereits eine If-Anweisung, die einen Kontext mit zwei Variablenumgebungen für die Evaluierung benötigt.

Im ersten Schritt wird mit Hilfe der Funktion θ ein Kontext für die Evaluierungsfunktion pval geschaffen.

$$\mathrm{env}_0 = \left\{(x, \underline{x}), (y, \underline{y})\right\}$$
$$\mathrm{ctx}_0 = \{(\mathrm{env}_0, \mathit{true}\}$$

Der Anfangskontext enthält nur eine Variablenumgebung env_0. Die Pfadbedingung der Variablenumgebung ist *true*. D.h., die Umgebung ist immer gültig. Die erste Anweisung ist die Hintereinanderausführung. Sie wird mit Hilfe der Formel 3.35 entwickelt.

$$\mathrm{pval}(\mathit{if}\ x < y\ \mathit{then}\ x := y - x\ \mathit{else}\ y := x - y; x := x + y, \mathrm{ctx}_0) =$$
$$\mathrm{pval}(x := x + y, \mathrm{pval}(\mathit{if}\ x < y\ \mathit{then}\ x := y - x\ \mathit{else}\ y := x - y, \mathrm{ctx}_0))$$

Die If-Anweisung ist jetzt separiert und wird mit der Formel 3.36 ausgewertet. Zunächst muß das Konditional mit allen Variablenumgebungen ausgeführt werden. Da nur eine symbolische Umgebung im Kontext vorhanden ist, brauchen wir nur eine Pfadbedingung berechnen. Für die Evaluierung des Konditionals $x < y$ wird

die Umgebung env_0 benötigt.

$$\begin{aligned} cond &= \text{sval}_C(x < y, \text{env}_0) = \\ &= \text{sval}_E(x, \left\{(x, \underline{x}), (y, \underline{y})\right\}) < \text{sval}_E(y, \left\{(x, \underline{x}), (y, \underline{y})\right\}) \\ &= \underline{x} < \underline{y} \end{aligned}$$

Die Pfadbedingung des Then-Zweigs ist

$$\begin{aligned} p_1 &= \mathit{true} \,\text{and}\, cond \\ &= cond \\ &= \underline{x} < \underline{y} \end{aligned}$$

Die Pfadbedingung kann aufgrund der Gesetze in der Funktionenalgebra (vgl. Abbildung 3.7) vereinfacht werden. Für den Else-Zweig wird das negierte, symbolisch ausgewertete Konditional der If-Anweisung herangezogen.

$$\begin{aligned} p_2 &= \mathit{true} \,\text{and not}\, cond \\ &= \text{not}(\underline{x} < \underline{y}) \\ &= \underline{x} \geq \underline{y} \end{aligned}$$

Die negierte Kleiner-Relation kann auch als Größergleich geschrieben werden (siehe Abbildung 3.7).

Beide Pfadbedinungen sind jetzt berechnet. Für die Auswertung beider Zweige müssen zwei Kontexte erzeugt werden. Der Kontext ctx_1 für den Then-Zweig enthält die Variablenumgebung env_0 mit der Pfadbedingung p_1. Für den Kontext ctx_2 des Else-Zweigs wird die Pfadbedingung p_2 verwendet.

$$\begin{aligned} \text{ctx}_1 &= \{(\text{env}_0, p_1)\} \\ \text{ctx}_2 &= \{(\text{env}_0, p_2)\} \end{aligned}$$

Die Auswertung des Then-Zweigs erfolgt mit dem Kontext ctx_1. Die Zuweisung $x := y - x$ wird nur mit einer Variablenumgebung env_0 evaluiert. Die Pfadbedinung p_1 der Variablenumgebung bleibt unverändert. Das Ergebnis des Then-Zweigs ist ein neuer Kontext ctx_3 mit nur einer Variablenumgebung und der da-

zugehörenden Pfadbedingung, die diese symbolische Umgebung erzwingt.

$$\begin{aligned}
\mathrm{ctx}_3 &= \mathrm{pval}(x := y - x, \mathrm{ctx}_1) \\
&= \mathrm{pval}(x := y - x, \{(\mathrm{env}_0, p_1)\}) \\
&= \{(\mathrm{sval}_S(x := y - x, env_0), p_1)\} \\
&= \left\{(\mathrm{sval}_S(x := y - x, \left\{(x, \underline{x}), (y, \underline{y})\right\}), p_1)\right\} \\
&= \left\{(\left\{(x, \underline{y} - \underline{x}), (y, \underline{y})\right\}, p_1)\right\}
\end{aligned}$$

Analog zur Auswertung des Then-Zweigs erfolgt die Evaluierung des Else-Zweigs. Die Berechnung wird mit dem Kontext ctx_2 durchgeführt.

$$\begin{aligned}
\mathrm{ctx}_4 &= \mathrm{pval}(y := x - y, \mathrm{ctx}_2) \\
&= \mathrm{pval}(y := x - y, \{(\mathrm{env}_0, p_2)\}) \\
&= \{(\mathrm{sval}_S(y := x - y, env_0), p_2)\} \\
&= \left\{(\mathrm{sval}_S(y := x - y, \left\{(x, \underline{x}), (y, \underline{y})\right\}), p_2)\right\} \\
&= \left\{(\left\{(x, \underline{x}), (y, \underline{x} - \underline{y})\right\}, p_2)\right\}
\end{aligned}$$

Beide Kontexte ctx_3 und ctx_4 werden zusammengefaßt. Die Vereinigung beider Mengen wird gebildet und das Ergebnis ctx_5 wird für die Auswertung des zweiten Statements in der Hintereinanderausführung verwendet.

$$\begin{aligned}
\mathrm{ctx}_5 &= \mathrm{ctx}_3 \cup \mathrm{ctx}_4 \\
&= \left\{\left(\left\{(x, \underline{y} - \underline{x}), (y, \underline{y})\right\}, p_1\right)\right\} \cup \left\{\left(\left\{(x, \underline{x}), (y, \underline{x} - \underline{y})\right\}, p_2\right)\right\} \\
&= \left\{\left(\left\{(x, \underline{y} - \underline{x}), (y, \underline{y})\right\}, p_1\right), \left(\left\{(x, \underline{x}), (y, \underline{x} - \underline{y})\right\}, p_2\right)\right\}
\end{aligned}$$

Anstatt der If-Anweisung wird der Kontext ctx_5 eingesetzt und die letzte Zuweisung ausgewertet. Da nun zwei mögliche Variablenumgebungen im ctx_5 vorhanden

sind, muß das letzte Statement mit beiden ausgewertet werden.

$$
\begin{aligned}
&\mathrm{ctx}_6 = \\
&\mathrm{pval}(x := x + y, \mathrm{pval}(\mathit{if}\ x < y\ \mathit{then}\ x := y - x\ \mathit{else}\ y := x - y, \mathrm{ctx}_0)) = \\
&\mathrm{pval}(x := x + y, \mathrm{ctx}_5) = \\
&\mathrm{pval}(x := x + y, \{\left(\left\{(x, \underline{y} - \underline{x}), (y, \underline{y})\right\}, p_1\right), \\
&\qquad \left(\left\{(x, \underline{x}), (y, \underline{x} - \underline{y})\right\}, p_2\right)\}) = \\
&\{\left(\mathrm{sval}_S(x := x + y, \left\{(x, \underline{y} - \underline{x}), (y, \underline{y})\right\}), p_1\right), \\
&\left(\mathrm{sval}_S(x := x + y, \left\{(x, \underline{x}), (y, \underline{x} - \underline{y})\right\}), p_2\right)\} = \\
&\{\left(\left\{(x, \underline{y} - \underline{x} + \underline{y}), (y, \underline{y})\right\}, p_1\right), \\
&\left(\left\{(x, \underline{x} + \underline{x} - \underline{y}), (y, \underline{x} - \underline{y})\right\}, p_2\right)\} =
\end{aligned}
$$

Das Ergebnis der symbolischen Auswertung des Beispiels 3.9 ist der Kontext ctx_6. Die Pfadbedingungen p_1 bzw. p_2 erzwingen die jeweilige symbolische Umgebung. Beide Bedingungen sind disjunkt.

Der Ausgabevektor wird mit der Funktion θ^{-1} erzeugt. Da zwei Variablenumgebungen im Kontext ctx_6 vorhanden sind, gibt es für jede Komponente der Ausgabe eine Fallunterscheidung mit zwei Fällen.

$$
\begin{aligned}
P(\underline{x}, \underline{y}) &= \theta^{-1}(\mathrm{ctx}_6) \\
&= \left(\begin{cases} \underline{y} - \underline{x} + \underline{y}, & \underline{x} < \underline{y} \\ \underline{x} + \underline{x} - \underline{y}, & \underline{x} \geq \underline{y} \end{cases}, \begin{cases} \underline{y}, & \underline{x} < \underline{y} \\ \underline{x} - \underline{y}, & \underline{x} \geq \underline{y} \end{cases}\right)
\end{aligned}
$$

Durch die Anwendung der Gesetze (vgl. Abbildung 3.7) kann das Ergebnis vereinfacht werden.

$$
P(\underline{x}, \underline{y}) = \left(\begin{cases} 2 * \underline{y} - \underline{x}, & \underline{x} < \underline{y} \\ 2 * \underline{x} - \underline{y}, & \underline{x} \geq \underline{y} \end{cases}, \begin{cases} \underline{y}, & \underline{x} < \underline{y} \\ \underline{x} - \underline{y}, & \underline{x} \geq \underline{y} \end{cases}\right)
$$

Optimierung

Die Pfad-enumerierende Methode erzeugt für jeden möglichen Pfad eine Variablenumgebung mit ihrer Pfadbedingung. Wenn ein geeignetes symbolisches Algebraprogramm zeigen kann, daß eine Pfadbedingung nicht erfüllbar ist, kann die Variablenumgebung aus dem Kontext herausgenommen werden. Je früher

eine Kontradiktion erkannt wird, desto weniger Umgebungen erzeugt die Evaluierungsfunktion pval. Eine If-Anweisung verdoppelt im schlechtesten Fall die im Kontext liegende Umgebung, d.h., daß die Daten exponentiell anwachsen.

Eine Anweisung verändert meistens nur einen Wert einer Variable in der Variablenumgebung. Es ist platzsparender, die Änderungen festzuhalten, und nicht die komplette Variablenumgebung neu zu berechnen, d.h., daß die Funktion vset nicht sofort ausgewertet wird[6].

3.4.2 If-Anweisungen in der Wert-konditionierenden Methode

Die symbolischen Werte der Variablen können effizienter dargestellt werden, wenn die Bedingungen direkt einer Variable zugeordnet werden (vgl. [13]). Eine Variable besitzt mehrere symbolische Werte und zu je einem Wert existiert eine Bedingung, die entscheidet, ob der Wert verwendet wird. Die Bedingungen müssen disjunkt sein und den Eingaberaum vollständig partitionieren, d.h., es muß für eine beliebige Parameterbelegung genau eine Bedingung zu *true* evaluieren. Die Bedingungen sind keine Pfadbedingungen, sondern klassifizieren Pfade.

Für die Evaluierung wird ein Kontext benötigt. Der Kontext wird analog zur Variablenumgebung definiert. Er ist eine endliche Menge aus Paaren. Jedes Paar besteht aus einem Variablensymbol und einer Fallunterscheidung. Die Fallunterscheidung ist eine endliche Menge aus Paaren. Das erste Element im Paar ist ein Ausdruck in $\mathbb{F}_E$ — das zweite ein Konditional in $\mathbb{F}_C$.

$$G = (\mathbb{F}_E \times \mathbb{F}_C)^l \tag{3.39}$$
$$\mathrm{env}_S \in (\mathrm{VS} \times G)$$

Die Menge G ist die Menge aller Fallunterscheidungen[7].

$$\{(e_1, c_1), \ldots, (e_l, c_l)\} \Leftrightarrow \begin{cases} e_1, & c_1 \\ \vdots & \\ e_l, & c_l \end{cases} \tag{3.40}$$

Die Fallunterscheidung wird in der hier vorgestellten Evaluierungsfunktion in zwei verschiedenen Darstellungen verwendet. Die erste Schreibweise ist die Mengennotation, die zweite die mathematische Notation.

---

[6] Für die Funktion vset wird eine "lazy evaluation"-Semantik verwendet.

[7] G enthält auch Fallunterscheidungen, die nicht gültig sind. Die symboliche Evaluierung muß daher garantieren, daß sie keine Fallunterscheidung erzeugt, deren Bedingungen nicht disjunkt bzw. den Eingaberaum nicht vollständig partitionieren.

Drei Hilfsfunktionen werden für die Bearbeitung einer Fallunterscheidung benötigt.

$$\alpha_i \left(\begin{Bmatrix} e_1, & c_1 \\ \vdots & \\ e_l, & c_l \end{Bmatrix} \right) = e_i, \quad \beta_i \left(\begin{Bmatrix} e_1, & c_1 \\ \vdots & \\ e_l, & c_l \end{Bmatrix} \right) = c_i, \quad \left| \begin{Bmatrix} e_1, & c_1 \\ \vdots & \\ e_l, & c_l \end{Bmatrix} \right| = l \tag{3.41}$$

Die Funktion α_i selektiert den i-ten Ausdruck in der Fallunterscheidung. β_i gibt die Bedingung des i-ten Ausdrucks zurück. Die Betragsfunktion der Fallunterscheidung bestimmt die Anzahl der Fälle.

Analog zur Pfad-enumerierenden Methode wird eine Evaluierungsfunktion cval definiert.

$$\text{cval} : S \times \text{CTX} \rightarrow \text{CTX} \tag{3.42}$$

Die Hintereinanderausführung und die Null-Anweisung unterscheiden sich zu den bisherigen Definitionen nicht.

$$\text{cval}(\mathit{null}, \text{ctx}) \mapsto \text{ctx} \tag{3.43}$$

$$\text{cval}(S_1; S_2, \text{ctx}) \mapsto \text{cval}(S_2, \text{cval}(S_1, \text{ctx})) \tag{3.44}$$

Im Ausdruck E einer Zuweisung $\mathit{var} := E$ werden Variablen verwendet, für die es mehr als einen Variablenwert gibt. Jeder Wert einer Variable, die in E vorkommt, ist möglich. Eine Kombination von möglichen Variablenwerten heißt Instanz.

Mit einer Instanz kann ein neuer Wert für die Variable *var* berechnet werden. Die Bedingung des neuen Wertes, wird von den Bedingungen der eingesetzten Werte der Variablen in E abgeleitet.

Die Funktion USE bestimmt die verwendeten Variablen in E.

$$\mathit{USE}(\mathit{expr}) = \{v_1, \ldots, v_m\} \tag{3.45}$$

Die Fallunterscheidungen der Variablen $v_1, \ldots, v_m$ werden aus dem Kontext ctx herausgelöst und im Vektor W festgehalten.

$$W = (\text{vget}(\text{ctx}, v_1), \ldots, \text{vget}(\text{ctx}, v_m)) \tag{3.46}$$

$$= (w_1, \ldots, w_m) \tag{3.47}$$

Die Funktion vget wird hier nicht explizit definiert, in der Formel 3.4 für die Standardsemantik von τ-Simple wurde sie bereits angegeben. Die Funktion vget

hat als Argumente den Kontext ctx und eine Variable v. Das Resultat der Funktion ist die Fallunterscheidung der Variable v.

Die möglichen Werte einer Variable v_k werden durchnumeriert und ein Indexwertebereich $\mathcal{I}_k$ bestimmt. $\mathcal{I}_k$ sind alle Zahlen von 1 bis zu der Anzahl der Fälle der Variable v_k.

$$\mathcal{I}_k = \{1, \ldots, |w_k|\} \tag{3.48}$$

Eine Instanz ist ein Element aus dem Kreuzprodukt aller Indexwertebereiche.

$$\mathcal{I} = \mathcal{I}_1 \times \cdots \times \mathcal{I}_m \tag{3.49}$$

Ein Element in $\mathcal{I}$ enthält für jede Variable eine Komponente. Die Komponente bestimmt den Wert für die Einsetzung in E.

$$I = (i_1, \ldots, i_m) \in \mathcal{I} \tag{3.50}$$

Der Indexvektorbereich kann auch leer sein. Dieser Fall tritt auf, wenn keine Variablen im Ausdruck vorkommen. Z.Bsp.: $x := 10$. Bei einem nicht trivialen Fall[8] wird für jede Instanz eine Variablenumgebung mit der dazugehörigen Bedingung erzeugt.

$$U = \{(\{(v_1, \alpha_{i_1}(w_1)), \ldots, (v_m, \alpha_{i_m}(w_m))\}, \beta_{i_1}(w_1) \,\text{and}\, \ldots \beta_{i_m}(w_m)) | \, I \in \mathcal{I}\} \tag{3.51}$$

Die Menge U besteht aus Paaren. Jedes Paar enthält eine Variablenumgebung und eine Bedingung einer Instanz. Die Bedingung der Instanz ist aus den Bedingungen der Werte der Variablen $v_1, \ldots, v_m$ zusammengesetzt.

Mit jeder Variablenumgebung aus U wird die Zuweisung $var := E$ ausgewertet, das Ergebnis mit der Bedingung der Umgebung gebunden und in die Fallunterscheidung der Variable var aufgenommen:

$$g = \{(\text{vget}\,(\text{sval}_S(var := E, \text{env}), p)) | \, (\text{env}, p) \in U\} \tag{3.52}$$

Wobei g die neue Fallunterscheidung der Variable var ist. Die Funktion vset ordnet der Variable var die Fallunterscheidung g zu[9].

$$\text{cval}(var := E, ctx) \mapsto \text{vset}(\text{ctx}, v, g) \tag{3.53}$$

[8] Der Indexvektorbereich ist nicht leer.

[9] vset wird analog zu Semantikfunktion val definiert. (Formel 3.5)

Für die If-Anweisung gibt es zwei verschiedene Formalismen. Der eine ist theoretisch einfacher, hat aber den Nachteil, daß bei einer Implementierung der Methode, Information zu spät bearbeitet wird. Aus didaktischen Gründen wird zunächst dieser Ansatz vorgestellt.

Analog zu der Auswertung einer Zuweisung in cval wird die Menge U für die Bedingung C der If-Anweisung $var := E$ berechnet.

Wenn

$$USE(C) = v_1, \ldots, v_m \tag{3.54}$$

die verwendeten Variablen im Konditional C sind, dann ist der Vektor W, der Indexvektorbereich $\mathcal{I}$ und die Menge durch

(3.55)
$$\begin{aligned} W &= (\mathrm{vget}(\mathrm{ctx}, v_1), \ldots, \mathrm{vget}(\mathrm{ctx}, v_m)) \\ &= (w_1, \ldots, w_m) \end{aligned}$$

(3.56)
$$\mathcal{I} = \{1, \ldots, |w_1|\} \times \cdots \times \{1, \ldots, |w_m|\}$$

(3.57)
$$U = \{(\{(v_1, \alpha_{i_1}(w_1)), \ldots, (v_m, \alpha_{i_m}(w_m))\}, \beta_{i_1}(w_1) \,\mathrm{and} \ldots \beta_{i_m}(w_m)) | \, I \in \mathcal{I}\}$$

bestimmt.

Das Konditional C wird mit jeder in U enthaltenen Variablenumgebung evaluiert und mit der dazugehörenden Bedingung verknüpft.

$$Y = \{\mathrm{sval}(C, \mathrm{env}) \,\mathrm{and}\, p | \, (\mathrm{env}, p) \in U\} \tag{3.58}$$

Jedes Element in Y ist ein Prädikat. Wie aus der nachfolgenden Beweisskizze folgt, ist genau ein Prädikat aus der Menge Y *true*, wenn der Then-Zweig in der If-Anweisung ausgeführt wird.

Die Disjunktion aller Elemente aus Y ergibt das Prädikat P_t, das genau dann *true* ist, wenn der Then-Zweig evaluiert wird.

$$P_t = \underset{y \in Y}{\mathrm{OR}}\, y \tag{3.59}$$

Analog kann das Prädikat P_f für den Else-Zweig entwickelt werden.

$$\overline{Y} = \{\mathrm{not}\, \mathrm{sval}(c, \mathrm{env}) \,\mathrm{and}\, p | \, (\mathrm{env}, p) \in U\} \tag{3.60}$$

Die Disjunktion aller Elemente in $\overline{y}$ ist ein Prädikat über den Parametern, das genau dann zu *true* evaluiert, wenn der Else-Zweig ausgeführt wird.

$$P_f = \underset{\overline{y} \in \overline{Y}}{\mathrm{OR}} \overline{y} \tag{3.61}$$

Die Evaluierungsfunktion cval wertet beide Pfade aus.

$$ctx_1 = \mathrm{sval}(S_1, \mathrm{ctx}) \tag{3.62}$$

$$ctx_2 = \mathrm{sval}(S_2, \mathrm{ctx}) \tag{3.63}$$

Die Hilfsfunktion Ext erweitert jede Bedingung einer Fallunterscheidung mit dem Prädikat P_f bzw. P_t.

(3.64)

$$\begin{aligned} \mathrm{Ext} :& G \times \mathbb{F}_C \to G \\ & \mathrm{Ext}(\{(e_1, c_1), \ldots, (e_l, c_l)\}, p) \mapsto \{(e_1, c_1 \,\mathrm{and}\, p), \ldots, (e_l, c_l \,\mathrm{and}\, p)\} \end{aligned}$$

Die Funktion Ext wird auf jede Fallunterscheidung in den Kontexten ctx_1 und ctx_2 angewendet. Das Ergebnis sind zwei erweiterte Kontexte ctx'_1 und ctx'_2.

$$\mathrm{ctx}'_1 = \{(v_1, \mathrm{Ext}(w_1, P_t)), \ldots, (v_n, \mathrm{Ext}(w_1, P_t))\} \tag{3.65}$$

$$\mathrm{ctx}'_2 = \{(v_1, \mathrm{Ext}(w_1, P_f)), \ldots, (v_n, \mathrm{Ext}(w_1, P_f))\} \tag{3.66}$$

Beide Kontexte werden im letzten Schritt zusammengefaßt. Eine weitere Hilfsfunktion Join verbindet Kontext ctx'_1 und Kontext ctx'_2.

$$\mathrm{Join} : \mathrm{CTX} \times \mathrm{CTX} \to \mathrm{CTX} \tag{3.67}$$

$$\begin{aligned} \mathrm{Join}(&\{(v_1, w_1), \ldots, (v_n, w_n)\}, \\ &\{(v_1, w'_1), \ldots, (v_n, w'_n)\}) \mapsto \\ &\{(v_1, w_1 \cup w'_1), \ldots, (v_n, w_n \cup w'_n)\} \end{aligned} \tag{3.68}$$

Der Kontext der evaluierten If-Anweisung ist daher

$$\mathrm{sval}(\mathit{if}\ C\ \mathit{then}\ S_1\ \mathit{else}\ S_2, \mathrm{ctx}) \mapsto \mathrm{Join}(\mathrm{ctx}'_1, \mathrm{ctx}'_2). \tag{3.69}$$

Die beiden Prädikate P_t und P_f sind disjunkt. Es kann daher keine ungültige Fallunterscheidung entstehen.

Beweisskizze

Zunächst muß gezeigt werden, daß die beiden Bedingungen P_t und P_f disjunkt sind. Das Prädikat $(P_t \,\text{and}\, P_f)$ muß in jedem beliebigen Kontext *false* sein. Es ist zu zeigen:

$$P_t \,\text{and}\, P_f = \mathit{false} \tag{3.70}$$

Zunächst setzen wir für P_t und P_f die Definition ein.

$$\left[\underset{y \in Y}{\text{OR}}\, y\right] \,\text{and}\, \left[\underset{\overline{y} \in \overline{Y}}{\text{OR}}\, \overline{y}\right] \tag{3.71}$$

Für jedes Element in Y bzw. $\overline{Y}$ wird folgende Notation verwendet.

$$y_i = (c_i \,\text{and}\, p_i) \tag{3.72}$$

$$\overline{y_i} = (\text{not}\, c_i \,\text{and}\, p_i) \tag{3.73}$$

Dabei ist c_i vom i-ten Element in der Menge Y bzw. $\overline{Y}$ das evaluierte Konditional und p_i ist die ererbte Bedingung der Instanz. Die Kardinaltität der Menge Y bzw. $\overline{Y}$ ist $|\mathcal{I}|$[†].

$$\left[\underset{1 \le i \le |\mathcal{I}|}{\text{OR}} (c_i \,\text{and}\, p_i)\right] \,\text{and}\, \left[\underset{1 \le j \le |\mathcal{I}|}{\text{OR}} (\text{not}\, c_j \,\text{and}\, p_j)\right] \tag{3.74}$$

Das Distributivgesetz wird angewendet, und der zweite Term der Und-Operation in den ersten Term hineingezogen.

$$\underset{1 \le i \le |\mathcal{I}|}{\text{OR}} \left[(c_i \,\text{and}\, p_i) \,\text{and}\, \underset{1 \le j \le |\mathcal{I}|}{\text{OR}} (\text{not}\, c_j \,\text{and}\, p_j)\right] \tag{3.75}$$

Das Distributivgesetz wird wieder verwendet, um den inneren Term der äußeren Oder-Operation zu vereinfachen.

$$\underset{1 \le i \le |\mathcal{I}|}{\text{OR}} \underset{1 \le j \le |\mathcal{I}|}{\text{OR}} [(\text{not}\, c_j \,\text{and}\, p_j) \,\text{and} (c_i \,\text{and}\, p_i)] = \tag{3.76}$$

$$\underset{1 \le i \le |\mathcal{I}|}{\text{OR}} \underset{1 \le j \le |\mathcal{I}|}{\text{OR}} (\text{not}\, c_j \,\text{and}\, c_i \,\text{and}\, p_i \,\text{and}\, p_j) \tag{3.77}$$

Wenn beide Indizes gleich sind, ergibt der innerste Term *false* und kann aus dem Indexbereich herausgenommen werden.

$$\underset{1 \le i \le |\mathcal{I}|}{\text{OR}} \underset{1 \le j \le |\mathcal{I}| \wedge i \neq j}{\text{OR}} (\text{not}\, c_j \,\text{and}\, c_i \,\text{and}\, p_i \,\text{and}\, p_j) \tag{3.78}$$

[†]Ist die Anzahl der Instanzen

p_i und p_j sind ererbte Bedingungen von unterschiedlichen Instanzen und haben folgende Struktur:

$$\underset{1\leq k\leq m}{\text{AND}}\ \beta_{i_k}(w_k) \tag{3.79}$$

Für p_i gibt es genau einen Indexvektor I_1 in $\mathcal{I}$ und für p_j den Indexvektor I_2. Beide Indexvektoren sind ungleich ($I_1 \neq I_2$), da p_i und p_j unterschiedliche Instanzen sind.

$$I_1 = (i_1^{I_1}, \ldots, i_m^{I_1}) \tag{3.80}$$

$$I_2 = (i_1^{I_2}, \ldots, i_m^{I_2}) \tag{3.81}$$

D.h, es existiert zumindest eine Komponente i_k im Vektor I_1, die sich von der gleichen Komponente im Vektor I_2 unterscheidet.

$$\exists k : i_k^{I_1} \neq i_k^{I_2} \wedge 1 \leq k \leq m \tag{3.82}$$

D.h., daß ein anderer Fall für die Instanz verwendet wurde.

$$p_i \,\text{and}\, p_j = \tag{3.83}$$

$$(\beta_{i_1^{I_1}}(w_1) \,\text{and}\, \ldots \beta_{i_k^{I_1}}(w_k) \,\text{and}\, \ldots \beta_{i_m^{I_1}}(w_m)) \,\text{and} \tag{3.84}$$

$$(\beta_{i_1^{I_2}}(w_1) \,\text{and}\, \ldots \beta_{i_k^{I_2}}(w_k) \,\text{and}\, \ldots \beta_{i_m^{I_2}}(w_m)) = \tag{3.85}$$

$$\mathit{false} \tag{3.86}$$

Da alle Bedingungen der Fälle disjunkt sein müssen, ist die Konjunktion von zwei Bedingungen einer Fallunterscheidung *false* und daher ist $p_i \,\text{and}\, p_j$ für ein $i \neq j$ immer *false*. Wenn $p_i \,\text{and}\, p_j = \mathit{false}$, dann ist $P_t \,\text{and}\, P_f = \mathit{false}$.

Jetzt muß nur noch gezeigt werden, daß nie eine ungültige Fallunterscheidung entstehen kann. Ausgehend von der Funktion θ, die nur einen Fall im Anfangskontext erzeugt, wird durch Induktion gezeigt, daß die If-Anweisung nur gültige Fallunterscheidungen erzeugen kann, wenn vor der If-Anweisung nur gültige Fallunterscheidungen waren. q.e.d.

Beispiel. Das Beispiel in Abbildung 3.9 wird mit der Wert-konditionierenden Methode symbolisch evaluiert. Die θ-Funktion erzeugt einen Anfangskontext.

$$\text{ctx}_0 = \Big\{(x, \{(\underline{x}, \mathit{true})\}), (y, \{(\underline{y}, \mathit{true})\})\Big\}$$

Die Fallunterscheidungen für die Variablen x und y sind in der Mengennotation angegeben. Sie beinhalten nur ein Paar. Die Bedingung ist daher *true*.

Die Hintereinanderausführung ist die erste Anweisung, die evaluiert wird.

$$\begin{aligned}&\mathrm{cval}(\mathit{if}\ x < y\ \mathit{then}\ x := y - x\ \mathit{else}\ y := x - y; x := x + y, \mathrm{ctx}_0) = \\ &\mathrm{cval}(x := x + y, \mathrm{cval}(\mathit{if}\ x < y\ \mathit{then}\ x := y - x\ \mathit{else}\ y := x - y, \mathrm{ctx}_0))\end{aligned} \tag{3.87}$$

Als nächstes wird die If-Anweisung ausgewertet. Die Prädikate P_t und P_f werden bestimmt.

$$\mathit{USE}(x < y) = \{x, y\}$$

Die USE-Menge der Bedingung $x < y$ beinhaltet die zwei Variablen x und y.

$$W = (\{(\underline{x}, \mathit{true})\}, \{(\underline{y}, \mathit{true})\})$$

Der Vektor W wird gebildet, indem die Fallunterscheidungen von x und von y aus dem Kontext ctx_0 herausgelöst werden.

$$\mathcal{I} = \{1\} \times \{1\}$$

Der Indexvektorbereich beinhaltet nur ein Element. D.h., es gibt für die symbolische Evaluierung der Bedingung nur eine Instanz.

$$\begin{aligned}U &= \{(\{(x, \underline{x}), (y, \underline{y})\}, \mathit{true}\ \mathrm{and}\ \mathit{true})\} \\ &= \{(\{(x, \underline{x}), (y, \underline{y})\}, \mathit{true})\}\end{aligned}$$

Die Menge U setzt sich aus den Variablenumgebungen der einzelnen Instanzen und deren ererbten Bedingungen zusammen. Da es nur eine Instanz gibt, besteht die Menge U aus einem Paar. Die Bedingung der Variablenumgebung ist *true*.

$$\begin{aligned}Y &= \{\mathrm{sval}_C(x < y, \{(x, \underline{x}), (y, \underline{y})\})\ \mathrm{and}\ \mathit{true}\} \\ &= \{\underline{x} < \underline{y}\ \mathrm{and}\ \mathit{true}\} \\ &= \{\underline{x} < \underline{y}\}\end{aligned}$$

Die Menge Y enthält die Prädikate der einzelnen Instanzen. Es ist genau dann ein Prädikat aus der Menge *true*, wenn der Then-Zweig ausgeführt wird.

Analog wird die Menge $\overline{Y}$ berechnet.

$$\begin{aligned}\overline{Y} &= \{\text{not}\,\text{sval}_C(x < y, \{(x, \underline{x}), (y, \underline{y})\})\,\text{and}\,\mathit{true}\} \\ &= \{\text{not}(\underline{x} < \underline{y})\,\text{and}\,\mathit{true}\} \\ &= \{\underline{x} \geq \underline{y}\}\end{aligned}$$

Da beide Mengen Y und $\overline{Y}$ aus nur einem Element bestehen, können P_t und P_f direkt angegeben werden. Würden mehrere Elemente in Y bzw. $\overline{Y}$ existieren, müßten die Prädikate mit der Oder-Operation verknüpft werde.

$$\begin{aligned}P_t &= \underline{x} < \underline{y} \\ P_f &= \underline{x} \geq \underline{y}.\end{aligned}$$

Im nächsten Schritt wird der Then-Zweig mit dem Kontext ctx_0 ausgewertet.

$$\text{ctx}_1 = \text{cval}(x := y - x, \text{ctx}_0)$$

Die Zuweisung $x := y - x$ verwendet im Ausdruck die zwei Variablen x und y. Da im Kontext jeweils ein Wert für beide Variablen gespeichert ist, existiert nur eine Instanz. Die ererbte Bedingung ist *true*. Analog zur Auswertung des Konditionals in der If-Anweisung, müssen die Hilfsmengen bestimmt werden.

$$\begin{aligned}\mathit{USE}(y - x) &= \{x, y\} \\ W &= (\{(\underline{x}, \mathit{true})\}, \{(\underline{y}, \mathit{true})\}) \\ \mathcal{I} &= \{1\} \times \{1\} \\ U &= \{(\{(x, \underline{x}), (y, \underline{y})\}, \mathit{true}\,\text{and}\,\mathit{true})\} \\ &= \{(\{(x, \underline{x}), (y, \underline{y})\}, \mathit{true})\}\end{aligned}$$

Die Menge U enthält nur ein Paar. Das erste Element des Paars ist die Variablenumgebung der Instanz, das zweite die ererbte Bedingung.

$$\begin{aligned}w &= \{(\text{vget}(\text{sval}_S(x := y - x, \{(x, \underline{x}), (y, \underline{y})\}), x), \mathit{true})\} \\ &= \{(\text{vget}(\{(x, \underline{y} - \underline{x}), (y, \underline{y})), \mathit{true})\} \\ &= \{(\underline{y} - \underline{x}, \mathit{true})\}\end{aligned}$$

Die Zuweisung wird mit der einen Variablenumgebung ausgewertet und der neue Wert der Variable x mit der Funktion vget aus der Ergebnisumgebung der Funk-

tion sval_S herausgeholt.

$$\begin{aligned}\text{ctx}_1 &= \text{vset}(\text{ctx}_0, x, \{(\underline{y} - \underline{x}, \mathit{true})\}) \\ &= \left\{(x, \{(\underline{y} - \underline{x}, \mathit{true})\}), (y, \{(\underline{y}, \mathit{true})\})\right\}\end{aligned}$$

Analog zum Then-Zweig wird der Else-Zweig berechnet. Die einzelnen Schritte werden aus Platzgründen nicht angegeben. Das Ergebnis ist

$$\begin{aligned}\text{ctx}_2 &= \text{cval}(x := y - x, \text{ctx}_0) \\ &= \left\{(x, \{(\underline{x}, \mathit{true})\}), (y, \{(\underline{x} - \underline{y}, \mathit{true})\})\right\}.\end{aligned}$$

Die Funktion Ext erweitert jede Bedingung im Kontext ctx_1 bzw. ctx_2 mit dem Prädikat P_f bzw. P_t.

$$\begin{aligned}\text{ctx}_1' &= \left\{(x, \text{Ext}(\{(\underline{y} - \underline{x}, \mathit{true})\}, P_t)), (y, \text{Ext}(\{(\underline{y}, \mathit{true})\}, P_t))\right\} \\ &= \left\{(x, \{(\underline{y} - \underline{x}, \underline{x} < \underline{y})\}), (y, \{(\underline{y}, \underline{x} \geq \underline{y})\})\right\}\end{aligned}$$

Die Bedingungen von ctx_2 werden mit dem Prädikat P_f erweitert.

$$\begin{aligned}\text{ctx}_2' &= \left\{(x, \text{Ext}(\{(\underline{x}, \mathit{true})\}, P_f)), (y, \text{Ext}(\{(\underline{x} - \underline{y}, \mathit{true})\}, P_f))\right\} \\ &= \left\{(x, \{(\underline{y}, \underline{x} \geq \underline{y})\}), (y, \{(\underline{x} - \underline{y}, \underline{x} \geq \underline{y})\})\right\}\end{aligned}$$

Die Funktion Join vereinigt beide Kontexte ctx_1' und ctx_2'. Das Ergebnis der Evaluierung der If-Anweisung ist ctx_3.

$$\begin{aligned}\text{ctx}_3 &= \text{Join}(ctx_1', ctx_2') \\ &= \{(x, \{(\underline{y} - \underline{x}, \underline{x} < \underline{y}), (\underline{x}, \underline{x} \geq \underline{y})\}), \\ &\quad (y, \{(\underline{y}, \underline{x} < \underline{y}), (\underline{x} - \underline{y}, \underline{x} \geq \underline{y})\})\} \\ &= \left\{\left(x, \begin{cases}\underline{y} - \underline{x}, & \underline{x} < \underline{y} \\ \underline{x}, & \underline{x} \geq \underline{y}\end{cases}\right), \left(y, \begin{cases}\underline{y}, & \underline{x} < \underline{y} \\ \underline{x} - \underline{y}, & \underline{x} \geq \underline{y}\end{cases}\right)\right\}\end{aligned}$$

Der Kontext ctx_3 wird in die Formel 3.87 eingesetzt und die letzte Anweisung ausgeführt.

(3.88)
$$\text{cval}(x := x + y, \text{cval}(\mathit{if}\ x < y\ \mathit{then}\ x := y - x\ \mathit{else}\ y := x - y, \text{ctx}_0)) =$$

(3.89)
$$\text{cval}(x := x + y, \text{ctx}_3)$$

Der Ausdruck $x + y$ verwendet die zwei Variablen x und y.

$$USE(x + y) = \{x, y\}$$

Der Vektor W faßt die zwei Fallunterscheidungen der beiden Variablen zusammen.

$$W = \Big(\{(\underline{y} - \underline{x}, \underline{x} < \underline{y}), (\underline{x}, \underline{x} \geq \underline{y})\}, \{(\underline{y}, \underline{x} < \underline{y}), (\underline{x} - \underline{y}, \underline{x} \geq \underline{y})\}\Big)$$

Für jede Variable gibt es zwei mögliche Werte. Der Indexvektorbereich besteht daher aus 4 Indexvektoren.

$$\begin{aligned} \mathcal{I} &= \{1, 2\} \times \{1, 2\} \\ &= \{(1, 1), (1, 2), (2, 1), (2, 2)\} \end{aligned}$$

Für jeden Indexvektor wird ein Paar in der Menge U erzeugt. Jedes Paar enthält eine Variablenumgebung der Instanz und deren Bedingung.

$$\begin{aligned} U &= \{u_1, u_2, u_3, u_4\} \\ u_1 &= \Big(\Big\{(x, \underline{y} - \underline{x}), (y, \underline{y})\Big\}, (\underline{x} < \underline{y}) \,\text{and}(\underline{x} < \underline{y})\Big) \\ u_2 &= \Big(\Big\{(x, \underline{y} - \underline{x}), (y, \underline{x} - \underline{y})\Big\}, (\underline{x} < \underline{y}) \,\text{and}(\underline{x} \geq \underline{y})\Big) \\ u_3 &= \Big(\Big\{(x, \underline{x}), (y, \underline{y})\Big\}, (\underline{x} \geq \underline{y}) \,\text{and}(\underline{x} < \underline{y})\Big) \\ u_4 &= \Big(\Big\{(x, \underline{x}), (y, \underline{x} - \underline{y})\Big\}, (\underline{x} \geq \underline{y}) \,\text{and}(\underline{x} \geq \underline{y})\Big) \end{aligned}$$

Die Bedingungen der Paare u_2 und u_3 sind nicht erfüllbar. u_2 und u_3 werden daher aus der Menge entfernt. Die Bedingungen von u_1 und u_4 enthalten Redundanzen.

$$\begin{aligned} U' &= \{u_1', u_4')\} \\ u_1' &= \Big(\Big\{(x, \underline{y} - \underline{x}), (y, \underline{y})\Big\}, \underline{x} < \underline{y}\Big) \\ u_4' &= \Big(\Big\{(x, \underline{x}), (y, \underline{x} - \underline{y})\Big\}, \underline{x} \geq \underline{y}\Big) \end{aligned}$$

Mit den Umgebungen in u_1 und u_4 wird die letzte Zuweisung ausgewertet, und die Bedingungen von u_1 und u_4 als Fallunterscheidung herangezogen.

$$
\begin{aligned}
w &= \{a_1, a_4\} \\
a_1 &= (\text{vget}(\text{sval}_S(x := x + y, \{(x, \underline{y} - \underline{x}), (y, \underline{y})\}), x), \underline{x} < \underline{y}) \\
&= (\underline{y} - \underline{x} + \underline{y}, \underline{x} < \underline{y}) \\
&= (2 * \underline{y} - \underline{x}, \underline{x} < \underline{y}) \\
a_4 &= (\text{vget}(\text{sval}_S(x := x + y, \{(x, \underline{x}), (y, \underline{x} - \underline{y})\}), x), \underline{x} \geq \underline{y}) \\
&= (\underline{x} + \underline{x} - \underline{y}, \underline{x} \geq \underline{y}) \\
&= (2 * \underline{x} - \underline{y}, \underline{x} \geq \underline{y})
\end{aligned}
$$

Die vset-Funktion trägt die neue Fallunterscheidung der Variable x in den Kontext ctx_4 ein.

$$
\begin{aligned}
\text{cval}(x := x + y, \text{ctx}_3) &= \text{vset}(\text{ctx}_3, x, w) \\
&= \{(x, \{(2 * \underline{y} - \underline{x}, \underline{x} < \underline{y}), (2 * \underline{x} - \underline{y}, \underline{x} \geq \underline{y})\}), \\
&\quad (y, \{(\underline{y}, \underline{x} < \underline{y}), (\underline{x} - \underline{y}, \underline{x} \geq \underline{y})\})\} \\
&= \left\{ \left(x, \begin{cases} 2 * \underline{y} - \underline{x}, & \underline{x} < \underline{y} \\ 2 * \underline{x} - \underline{y}, & \underline{x} \geq \underline{y} \end{cases} \right), \left(y, \begin{cases} \underline{y}, & \underline{x} < \underline{y} \\ \underline{x} - \underline{y}, & \underline{x} \geq \underline{y} \end{cases} \right) \right\} \\
&= \text{ctx}_4
\end{aligned}
$$

θ^{-1} bildet nur mehr die einzelnen Fallunterscheidungen der Variablen x und y auf die Komponenten des Ausgabevektors ab.

$$
\begin{aligned}
P(\underline{x}, \underline{y}) &= \theta^{-1}(\text{ctx}_4) \\
&= \left(\begin{cases} 2 * \underline{y} - \underline{x}, & \underline{x} < \underline{y} \\ 2 * \underline{x} - \underline{y}, & \underline{x} \geq \underline{y} \end{cases}, \begin{cases} \underline{y}, & \underline{x} < \underline{y} \\ \underline{x} - \underline{y}, & \underline{x} \geq \underline{y} \end{cases} \right)
\end{aligned}
$$

Optimierung und Vereinfachung

Vereinfachungen Die folgenden Vereinfachungen ergeben sich durch die Eigenschaften der Bedingungen in der Wert-konditionierenden Methode.

$$\begin{cases} \vdots & \\ e_{t-1} & , c_{t-1} \\ e_t & , true \\ e_{t+1} & , c_{t+1} \\ \vdots & \end{cases} \iff e_t \tag{3.90}$$

$$\begin{cases} \vdots & \\ e_{t-1} & , c_{t-1} \\ e_t & , false \\ e_{t+1} & , c_{t+1} \\ \vdots & \end{cases} \iff \begin{cases} \vdots & \\ e_{t-1} & , c_{t-1} \\ e_{t+1} & , c_{t+1} \\ \vdots & \end{cases} \tag{3.91}$$

$$\begin{cases} \vdots & \\ e_s & , c_s \\ \vdots & \\ e_s & , c_t \\ \vdots & \end{cases} \iff \begin{cases} \vdots & \\ e_s & , c_s \text{ or } c_t \\ \vdots & \end{cases} \tag{3.92}$$

If-Anweisung Die If-Anweisung *if C then* S_1 *else* S_2 verdoppelt die Werte in der Fallunterscheidung auch für jene Variablen, die weder im Then- noch im Else-Zweig in der DEF-Menge enthalten sind. Nach der Auswertung der If-Anweisung haben diese Variablen folgende Struktur:

$$\{\dots(v_i, \{(e_1, p_1), (e_1, p_1'), \dots, (e_l, p_l), (e_l, p_l')\})\dots\} \tag{3.93}$$

Die Semantik der Fallunterscheidung wird nicht verändert, wenn gleiche Werte zusammengefaßt werden. Die Bedingungen werden mit der Oder-Operation verknüpft.

$$\{\dots(v_i, \{(e_1, p_1 \text{ or } p_1'), \dots, (e_l, p_l \text{ or } p_l')\})\dots\} \tag{3.94}$$

Dir Prädikate p_j und p'_j haben aufgrund der Evaluierung folgende Form:

$$p_j = (c_j \text{ and } P_t) \tag{3.95}$$

$$p'_j = (c_j \text{ and } P_f) \tag{3.96}$$

Die Disjunktion beider Bedingungen ergibt c_j.

$$p_j \text{ or } p'_j = \tag{3.97}$$

$$(c_j \text{ and } P_t) \text{ or} (c_j \text{ and } P_f) = \tag{3.98}$$

$$c_j \text{ or} (P_t \text{ and } P_f) = \tag{3.99}$$

$$c_j \tag{3.100}$$

D.h., daß die If-Anweisung den symbolischen Wert einer Variable nicht verändert, wenn die Variable in keinem der beiden Zweige definiert wird. Um den obigen Vereinfachungsschritt in der Evaluierung zu berücksichtigen, kann die Evaluierungsfunktion diese Variablen gesondert behandeln.

Bedingungen P_t und P_f werden als Argument in cval mitgeführt. Wenn eine neue Fallunterscheidung für eine Variable erzeugt wird, kann die vollständige Bedingung mit einem Algebra-Programm überprüft werden, ob sie eine Kontradiktion oder eine Tautologie ist. Ein frühzeitiges Erkennen von redundanten Bedingungen verhindert das exponentielle Anwachsen von nutzloser Information.

Die Wert-konditionierende Methode erzeugt Instanzen, die aufgrund der Programmstruktur nicht auftreten können. Diese Fälle können mit einem zur Zeit noch in Entwicklung stehenden graphentheoretischen Ansatz schnell erkannt werden. Erfolgt die Überprüfung mit einem Algebra-Programm, so sind zeitaufwendige Algebra-Algorithmen notwendig, um redundante Bedingungen bzw. Kontradiktionen zu erkennen. Die Algorithmen haben zumindest quadratischen Aufwand. Die Daten sind dabei Bedingungen, die sehr schnell anwachsen können. D.h., daß für eine konkrete Implementierung der Wert-konditionierenden Methode ein effizientes Verfahren für die nicht durchführbaren Instanzen (im Beispiel u_2, u_4) entwickelt werden muß.

3.4.3 If-Anweisungen in der Logik-orientierten Methode

Die letzte hier vorgestellte Methode erzeugt aus einem imperativen Programm ein Logikprogramm[53, 41]. Wenn aufgrund einer If-Anweisung eine Variable zwei Werte besitzt, wird der Variable eine Logikvariable zugewiesen und in Abhängigkeit

der Pfadbedingung der Wert der Logikvariable gebunden. Dieser Formulismus wurde in [18, 15, 14, 2] vorgestellt.

Die Evaluierungsfunktion lval der Logik-orientierten Methode ordnet in einem beliebigen Punkt des Programms einer Variable einen symbolischen Ausdruck zu. Der Ausdruck ist Element der mit Logikvariablen erweiterten Funktionenalgebra $\mathbb{F}$. Der Kontext ctx für lval besteht aus einer symbolischen Variablenumgebung für die einzelnen Variablen und einer Pfadbedingung. Die Pfadbedingung bindet in Abhängigkeit der Parameter die einzelnen Logikvariablen.

Die Funktion lval wird analog zu den bisherigen symbolischen Evaluierungsfunktionen definiert.

$$\mathrm{lval} : S \times \mathrm{CTX} \to \mathrm{CTX} \tag{3.101}$$

Der Kontext ist ein Paar, das aus einer Variablenumgebung und einer Pfadbedingung aufgebaut ist.

$$\mathrm{CTX} = \mathrm{ENV}_S \times \mathbb{F}_C \tag{3.102}$$

Die Hintereinanderausführung, die Null-Anweisung und die Zuweisung haben eine ähnliche Definition wie in der Evaluierungsfunktion sval.

$$\mathrm{lval}(\mathit{null}, \mathrm{ctx}) \mapsto \mathrm{ctx} \tag{3.103}$$

$$\mathrm{lval}(S_1; S_2, \mathrm{ctx}) \mapsto \mathrm{lval}(S_2, \mathrm{lval}(S_1, \mathrm{ctx})) \tag{3.104}$$

$$\mathrm{lval}(\mathit{var} := E, (\mathrm{env}, p)) \mapsto (\mathrm{sval}_S(\mathit{var} := E, \mathrm{env}), p) \tag{3.105}$$

Die Zuweisung verändert die Pfadbedingung nicht.

Die If-Anweisung wertet beide Zweige aus. Ein Faltungsoperator $\circ$ fügt die zwei Kontexte der Zweige zusammen.

Einer Variable, für die mindestens eine Zuweisung in einem Zweig der If-Anweisung existiert, wird nach der If-Anweisung eine neue logische Variable zugeordnet. Der Wert der logischen Variable wird mit dem Wert des Then-Zweigs gebunden, falls die Bedingung der If-Anweisung *true* ergibt. Im anderen Fall wird der Wert des Else-Zweigs gebunden.

$$\begin{aligned} \mathrm{lval}(\mathit{if}\ C\ \mathit{then}\ S_1\ \mathit{else}\ S_2, (\mathrm{env}, p)) = \\ \mathrm{lval}(S_1, (\mathrm{env}, p \,\mathrm{and}\, \mathrm{sval}_C(C, \mathrm{env})) \circ \\ \mathrm{lval}(S_2, (\mathrm{env}, p \,\mathrm{and}\, \mathrm{not}(\mathrm{sval}_C(C, \mathrm{env})))) \end{aligned} \tag{3.106}$$

Der Faltungsoperator $\circ$ extrahiert jene Wertpaare im Kontext ctx_1 und ctx_2, die unterschiedlich sind.

$$\circ : \mathrm{CTX} \times \mathrm{CTX} \to \mathrm{CTX} \tag{3.107}$$

Wenn die Kontexte ctx_1 und ctx_2

$$\mathrm{ctx}_1 = \left(\{(v_1, w_1^{(1)}), \ldots, (v_n, w_n^{(1)})\}, p_1\right) \tag{3.108}$$

$$\mathrm{ctx}_2 = \left(\{(v_1, w_1^{(2)}), \ldots, (v_n, w_n^{(2)})\}, p_2\right) \tag{3.109}$$

sind, dann ist die neue Variablenumgebung env' der Faltungsoperation

$$\mathrm{env}' = \{(v_i, z_i) |\, 1 \leq i \leq n \wedge z_i = \begin{Bmatrix} w_i^{(1)}, & w_i^{(1)} = w_i^{(2)} \\ \ell_i, & sonst \end{Bmatrix} \tag{3.110}$$

ℓ_i ist eine neue logische Variable. Sie wird genau dann als Wert einer Variable verwendet, wenn der Then-Zweig einen anderen symbolischen Wert als der Else-Zweig besitzt.

$$\lambda_1 = \underset{1 \leq i \leq n \wedge w_i^{(1)} \neq w_i^{(2)}}{\mathrm{AND}} (\ell_i = w_i^{(1)}) \tag{3.111}$$

$$\lambda_2 = \underset{1 \leq i \leq n \wedge w_i^{(1)} \neq w_i^{(2)}}{\mathrm{AND}} (\ell_i = w_i^{(2)}) \tag{3.112}$$

λ_1 ist ein Prädikat und bindet die logischen Variablen mit den Werten des Then-Zweigs. Analog wird λ_2 für den Else-Zweig definiert. Das Ergebnis der Faltungsoperation ist ein neuer Kontext mit der Variablenumgebung env' und einer neuen Pfadbedingung.

$$\mathrm{ctx}_1 \circ \mathrm{ctx}_2 = (env', (p_1 \,\mathrm{and}\, \lambda_1) \,\mathrm{or} (p_2 \,\mathrm{and}\, \lambda_2)) \tag{3.113}$$

λ_1 bzw. λ_2 wird mit der Pfadbedingung des Then-Zweigs bzw. Else-Zweigs verknüpft. Die erweiterten Pfadbedingungen der beiden Zweige werden disjungiert.

Die Funktion θ wird analog zu den bisherigen Evaluierungsfunktionen definiert.

$$\theta(x_1, \ldots, x_n) \mapsto (\{(v_1, x_1), \ldots, (v_n, x_n)\}, true) \tag{3.114}$$

Die Umkehrfunktion θ^{-1} ist wesentlich komplexer. Aus dem Logikprogramm kann nicht direkt die funktionale Definition des Programms abgeleitet werden. Ein weiterer Bearbeitungsschritt ist notwendig, um die einzelnen Fälle aus dem Logikprogramm zu extrahieren.

3.4.4 Beispiel

Das Beispiel aus Abbildung 3.9 wird mit der Logik-orientierten Methode ausgewertet. Der Anfangskontext wird mit Hilfe der θ-Funktion erzeugt.

$$\mathrm{ctx}_0 = (\{(x, \underline{x}), (y, \underline{y})\}, \mathit{true})$$

Im nächsten Schritt wertet die Funktion lval die Hintereinanderausführung aus. Die Zuweisung und die If-Anweisung sind separiert.

$$\mathrm{lval}(\mathit{if}\ x < y\ \mathit{then}\ x := y - x\ \mathit{else}\ y := x - y; x := x + y, \mathrm{ctx}_0) =$$
$$\mathrm{lval}(x := x + y, \mathrm{lval}(\mathit{if}\ x < y\ \mathit{then}\ x := y - x\ \mathit{else}\ y := x - y, \mathrm{ctx}_0))$$

Die Bedingung der If-Anweisung wird mit ctx_0 ausgewertet.

$$\mathrm{sval}_C(x < y, \{(x, \underline{x}), (y, \underline{y})\}) =$$
$$\mathrm{sval}_E(x, \{(x, \underline{x}), (y, \underline{y})\}) < \mathrm{sval}_E(y, \{(x, \underline{x}), (y, \underline{y})\}) =$$
$$\underline{x} < \underline{y}$$

Die Pfadbedingung für den Then-Zweig ist die Konjunktion der evaluierten Bedingung und der bisherigen Pfadbedingung.

$$p_1 = \mathit{true}\ \mathrm{and} (\underline{x} < \underline{y})$$
$$= \underline{x} < \underline{y}$$

Für den Else-Zweig wird das evaluierte Konditional der If-Anweisung negiert.

$$p_2 = \mathit{true}\ \mathrm{and}\ \mathrm{not}(\underline{x} < \underline{y})$$
$$= \underline{x} \geq \underline{y}$$

Der Then-Zweig wird mit der Umgebung aus ctx_0 und mit der Pfadbedingung p_1 ausgewertet.

$$\mathrm{lval}(x := y - x, (\{(x, \underline{x}), (y, \underline{y})\}, p_1)) =$$
$$(\mathrm{sval}_S(x := y - x, \{(x, \underline{x}), (y, \underline{y})\}), p_1) =$$
$$(\{(x, \underline{y} - \underline{x}), (y, \underline{y})\}, p_1) = \mathrm{ctx}_1$$

Analog wird der Else-Zweig ausgewertet.

$$\mathrm{lval}(y := x - y, (\{(x, \underline{x}), (y, \underline{y})\}, p_2)) =$$
$$(\mathrm{sval}_S(y := x - y, \{(x, \underline{x}), (y, \underline{y})\}), p_2) =$$
$$(\{(x, \underline{x}), (y, \underline{x} - \underline{y})\}, p_2) = \mathrm{ctx}_2$$

Das Ergebnis der If-Anweisung ist die Faltung beider Kontexte ctx_1 und ctx_2.

$$\begin{aligned}\mathrm{ctx}_1 \circ \mathrm{ctx}_2 = \\ (\{(x, \underline{y} - \underline{x}), (y, \underline{y})\}, p_1) \circ (\{(x, \underline{x}), (y, \underline{x} - \underline{y})\}, p_2) = \\ (\{(x, \ell_1), (y, \ell_2)\}, p_3) = \mathrm{ctx}_3\end{aligned}$$

In beiden Zweigen haben x und y unterschiedliche Werte und müssen daher durch die logischen Variablen ℓ_1 und ℓ_2 ersetzt werden.

$$\begin{aligned}p_3 = (p_1 \,\mathrm{and}(\ell_1 = \underline{y} - \underline{x}) \,\mathrm{and}(\ell_2 = \underline{y})) \,\mathrm{or} \\ (p_2 \,\mathrm{and}(\ell_1 = \underline{x}) \,\mathrm{and}(\ell_2 = \underline{x} - \underline{y}))\end{aligned}$$

Die Pfadbedingung der Faltung besteht aus den beiden Pfadbedingungen p_1 und p_2 mit den einzelnen Variablenbedingungen der logischen Variablen.

$$\begin{gathered}\mathrm{lval}(x := x + y, \mathrm{lval}(\mathit{if}\ x < y\ \mathit{then}\ x := y - x\ \mathit{else}\ y := x - y, \mathrm{ctx}_0)) = \\ \mathrm{lval}(x := x + y, \mathrm{ctx}_3) = \\ (\mathrm{sval}_S(x := x + y, \{(x, \ell_1), (y, \ell_2)\}), p_3) = \\ (\{(x, \ell_1 + \ell_2), (y, \ell_2)\}, (p_1 \,\mathrm{and}(\ell_1 = \underline{y} - \underline{x}) \,\mathrm{and}(\ell_2 = \underline{y})) \,\mathrm{or} \\ (p_2 \,\mathrm{and}(\ell_1 = \underline{x}) \,\mathrm{and}(\ell_2 = \underline{x} - \underline{y})))\end{gathered}$$

Im letzten Schritt wird die zweite Anweisung der Hintereinanderausführung evaluiert. Das Ergebnis ist ein logisches Programm.

3.5 Schleifen

Die einzige Schleife in τ-Simple ist die While-Schleife. Sie überprüft zu Beginn der Schleife die Schleifenbedingung und führt nur dann den Schleifenrumpf aus, wenn die Schleifenbedingung *true* ist. Andernfalls wird mit der nächsten Anweisung fortgefahren.

Repeat-Schleifen oder komplexere Schleifen können mit Hilfe der While-Schleife formuliert werden. Die Semantikfunktion würde unnötig anwachsen, wenn sie in die Sprache aufgenommen würden.

$$\mathit{repeat}\ S\ \mathit{until}\ C \;\Leftrightarrow\; S; \mathit{while}\ \mathrm{not}\ C\ \mathit{do}\ S \tag{3.115}$$

Eine Repeat-Schleife führt den Schleifenrumpf mindestens einmal aus. Sie wird daher in eine While-Schleife umgewandelt, vor der S ausgeführt wird. Danach

folgt die While-Schleife mit der negierten Repeat-Schleifenbedingung und dem gleichen Schleifenrumpf.

For-Schleifen werden analog zur Repeat-Schleife mit Hilfe eines Schemas in While-Schleifen umgewandelt.

$$for\ v := a\ to\ e\ do\ S \Leftrightarrow v := a; while\ a \leq e\ do\ S; v := v + 1 \tag{3.116}$$

Die symbolische Analyse benötigt die Semantik der Schleife. val_S wird für die Auswertung der Schleife rekursiv definiert.

$$\mathrm{val}_S(while\ C\ do\ S, \mathrm{env}) \mapsto \tag{3.117}$$
$$\begin{cases} \mathrm{val}_S(while\ C\ do\ S, \mathrm{val}_S(S, \mathrm{env})), & \mathrm{val}_C(C, \mathrm{env}) = true \\ \mathrm{env}, & sonst \end{cases}$$

Der Schleifenrumpf S der Schleife wird solange evaluiert, bis die Schleifenbedingung C *false* ergibt.

Beispiel. Das Beispiel in Abbildung 3.10 wird ausgewertet. Die Anfangswerte von x und y können beliebig gewählt werden, da nach der zweiten Anweisung die beiden Variablen die Werte $x = 10$ und $y = 11$ besitzen. Wir beginnen also mit der Evaluierung der While-Schleife mit dem Kontext env_2[†].

$$\mathrm{env}_2 = \{(x, 10), (y, 11)\}$$
$$\mathrm{val}_S(while\ x < y\ do\ x := x + 1, \mathrm{env}_2)$$

Jetzt wird das Konditional der While-Schleife mit env_2 berechnet.

$$\begin{aligned} \mathrm{val}_C(x < y, \mathrm{env}_2) &= \mathrm{val}_E(x, \mathrm{env}_2) < \mathrm{val}_E(y, \mathrm{env}_2) \\ &= 10 < 11 \\ &= true \end{aligned}$$

Die Schleifenbedingung ist *true* und daher wird der Schleifenrumpf evaluiert.

$$\mathrm{val}_S(while\ x < y\ do\ x := x + 1, \mathrm{env}_2) =$$
$$\mathrm{val}_S(while\ x < y\ do\ x := x + 1, \mathrm{val}_S(x := x + 1, \mathrm{env}_2))$$

[†]Die Evaluierung der ersten beiden Anweisungen ist trivial.

Die Zuweisung im Schleifenrumpf erhöht den Wert der Variable x um 1.

$$\begin{aligned} \mathrm{val}_S(x := x + 1, \mathrm{env}_2) &= \{(x, 11), (y, 11)\} \\ &= \mathrm{env}_3 \end{aligned}$$

Die Schleifenbedingung wird mit env_3 ausgewertet. Das Ergebnis der Auswertung ist *false* und die Schleife terminiert.

$$\begin{aligned} \mathrm{val}_C(x < y, \mathrm{env}_3) &= \mathrm{val}_E(x, \mathrm{env}_3) < \mathrm{val}_E(y, \mathrm{env}_3) \\ &= 11 < 11 \\ &= \mathit{false} \end{aligned}$$

Das Resultat des Programms ist die Variablenumgebung env_3.

DEF/USE-Mengen

Die DEF-Menge der While-Schleife entspricht der DEF-Menge des Schleifenrumpfes S. Die USE-Menge der Schleife besteht aus den USE-Mengen der Schleifenbedingung C und des Schleifenrumpfes.

$$DEF(\mathit{while}\, C\, \mathit{do}\, S) = DEF(S) \tag{3.118}$$

$$USE(\mathit{while}\, C\, \mathit{do}\, S) = USE(S) \cup USE(C) \tag{3.119}$$

3.5.1 Symbolische Exekution der Schleife

Eine symbolisch entscheidbare Schleife kann mit einer beliebigen symbolischen Methode[10] aufgelöst werden, indem solange der Schleifenrumpf symbolisch evaluiert wird, bis die Schleifenbedingung für jede weitere Iteration *false* ist. Schleifen, die dieser Klasse angehören, sind selten und daher ist die Exekution einer Schleife in der Regel nicht möglich.

Das Beispiel in Abbildung 3.10 enthält eine symbolisch entscheidbare Schleife, da ohne die Eingabe zu kennen, die Schleife immer nur einmal durchlaufen wird. Eine symbolisch unentscheidbare Schleife entsteht, wenn die ersten zwei Zuweisungen im Programm Abbildung 3.10 entfernt werden. Die symbolische Evaluierungsfunktion kann bei jedem Durchlauf der Schleife nicht entscheiden, ob die Schleife terminiert oder nicht, d.h., ein unendlich langer Entscheidungsbaum würde entstehen.

[10] Pfad-enumerierend, Wert-konditionierend bzw. Logik-orientiert

```
decl x,y:[1..100]

x:= 10;
y:= 11;
while x < y
   x:=x+1
```

Abbildung 3.10: Beispiel einer symbolisch entscheidbaren Schleife

In der Pfad-enumerierenden Methode wird formal eine While-Schleife durch folgende Definition behandelt.

(3.120)

$$\begin{aligned}
&\mathrm{pval}(\mathit{while}\, C\, \mathit{do}\, S, \{(\mathrm{env}_1, p_1), \ldots, (\mathrm{env}_k, p_k)\}) \mapsto \\
&\qquad \mathrm{pval}(\mathit{while}\, C\, \mathit{do}\, S, \\
&\mathrm{pval}(S, \{(\mathrm{env}_1, p_1 \,\mathrm{and}\, \mathrm{sval}_C(C, \mathrm{env}_1)), \ldots, (\mathrm{env}_k, p_k \,\mathrm{and}\, \mathrm{sval}_C(C, \mathrm{env}_k))\})) \\
&\cup \{(\mathrm{env}_1, p_1 \,\mathrm{and}\, \mathrm{not}\, \mathrm{sval}_C(C, \mathrm{env}_1)), \ldots, (\mathrm{env}_k, p_k \,\mathrm{and}\, \mathrm{not}\, \mathrm{sval}_C(C, \mathrm{env}_k))\}
\end{aligned}$$

Beide Möglichkeiten — Schleife terminiert bzw. Schleife durchläuft den Schleifenrumpf — werden berücksichtigt. Falls die Schleife terminiert, wird die Pfadbedingung mit der negierten Schleifenbedingung verknüpft. Im anderen Fall wird rekursiv die Schleife evaluiert, und die Pfadbedingung mit der evaluierten Schleifenbedingung erweitert. Die Kontexte der beiden Fälle werden vereinigt und sind das Ergebnis der Evaluierung der While-Anweisung.

Die Abbildung 3.12 enthält die Kontexte der Evaluierung des Beispiels 3.11. Es entstehen bei der symbolischen Auswertung unendlich viele Pfade.

```
decl x,y:[1..100];

while x < y do
   x:=x+1
```

Abbildung 3.11: Beispiel einer symb. unentscheidbaren Schleife

Wenn Schleifen exekutiert werden und die Schleife gehört der Klasse der symbolisch unentscheidbaren Schleifen an, so kann vorerst keine funktionale Definition des Programms gefunden werden.

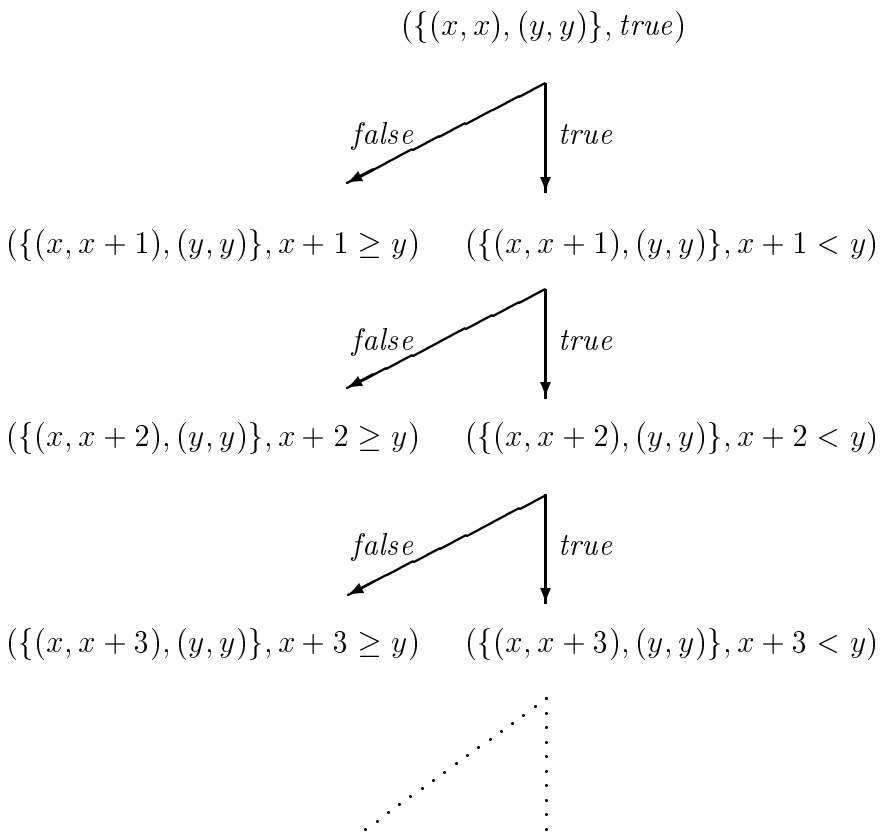


Abbildung 3.12: Evaluierung einer symbolisch unentscheidbaren Schleife

Im Compilerbau (vgl. [1]) wird diese Technik als "Aufrollen einer Schleife" bezeichnet. Die Konsequenz sind längere Programme. In der symbolischen Evaluierung wird eine symbolisch entscheidbare Schleife zur Gänze aufgerollt und der Kontext wächst exponentiell an. Es ist daher vorteilhaft, die symbolischen Ausdrücke zu vereinfachen. Basierend auf den in Abbildung 3.7 vorgestellten Rechenregeln kann mit einem symbolischen Algebraprogramm (vgl. [63]) die Vereinfachung durchgeführt werden. Gute Heuristiken sind notwendig, um die Ausdrücke schnell zu vereinfachen.

Symbolisch unentscheidbare Schleifen

Symbolisch unentscheidbare Schleifen lassen sich zu symbolisch entscheidbaren Schleifen transformieren. Die Bedingung für diese Transformation ist die Endlichkeit des Eingaberaums.

In Abhängigkeit von der Methode existieren Bedingungen, die angeben, in welchem Teilraum der Eingabe, die Schleife zur Ausführung kommt. Da der Ein-

gaberaum endlich ist[11], ist auch der Teilraum endlich, und mit jeder möglichen Eingabekonfiguration wird die Schleife symbolisch durchlaufen.

```
decl x:[1..100];

if x > 90 then
   while x < 100
     x := x + 1
else
   null
```

Abbildung 3.13: Beispiel einer symbolisch unentscheidbaren Schleife

Das Beispiel in Abbildung 3.13 enthält eine symbolisch unentscheidbare Schleife, da der Parameter $\underline{x}$ unbekannt ist. Wir wissen aufgrund der rg-Funktion, daß x nur Werte zwischen 1 und 100 annehmen kann, d.h., wir können daraus ein äquivalentes Programm mit entscheidbaren Schleifen erzeugen (Abbildung 3.14).

Das Beispiel aus Abbildung 3.13 wurde in ein Programm mit symbolisch entscheidbaren Schleifen transformiert. Der Teilraum der Eingabe, der die unentscheidbaren Schleifen sicher ausführen läßt, wird vollständig aufgezählt. Die Schleife wird damit entscheidbar.

In den wenigsten Fällen ist es notwendig, den vollständigen Teilraum der Eingabe aufzuzählen. Die Menge der Variablen, die für die Berechnung der Schleifenbedingung benötigt wird, ist meistens kleiner als die Menge aller Variablen, d.h., der Raum beschränkt sich auf einen Unterraum des Teilraums der Eingabe. Der Unterraum wird nur aus den Variablen aufgespannt, die zur Berechnung der Schleifenbedingung benötigt werden[12].

Um die benötigten Variablen für die Berechnung der Schleifenbedingung zu finden, reicht ein einfacher Markierungsalgorithmus (Abbildung 3.15). Er markiert die in der Bedingung vorkommenden Variablen. Danach werden die Variablen, die in der USE-Menge einer Anweisung enthalten sind, gekennzeichnet, falls die Anweisung eine bereits markierte Variable in der DEF-Menge enthält. Der Markierungsalgorithmus stoppt, wenn er keine weiteren Variablen findet. Die Menge M enthält am Ende alle markierten Variablen.

[11]Für jede Variable gibt die rg-Funktion den Wertebereich an.

[12]Die Variablen entscheiden während der Ausführung, ob die Schleife abgebrochen wird oder nicht.

```
decl x:[1..100];

if x > 90 then
    if x = 91 then
        while x < 100
            x := x + 1
    else if x = 92 then
        while x < 100
            x := x + 1
    else if x = 93 then
    .
    .
    .
    else if x = 100 then
        while x < 100
            x := x +1
else
    null
```

Abbildung 3.14: Transformiertes Programm

$M = USE(C)$
$M_n = \emptyset$
solange $M \neq M_n$
    $M_n = M$
    für jedes S_i in S
        wenn $DEF(S_i) \cap M \neq \emptyset$
            $M_n = M_n \cup USE(S_i)$
$M_n = M$

Abbildung 3.15: Markierungsalgorithmus für eine eine While-Schleife *while C do S*

Um den Teilraum zu finden, werden die Bedingungen, die die Schleife ausführen lassen, mit dem endlichen Raum der Eingabe geschnitten. Das Ergebnis ist wieder ein endlicher Raum. Für jedes Element im Unterraum des vorher bestimmten Teilraums wird eine If-Anweisung erzeugt. Die If-Anweisung enthält die Schleife (siehe Abbildung 3.16).

3.5.2 Schleifen und Fixpunkttheoreme

In [19] wurde die abstrakte Interpretation vorgestellt. In jedem Punkt des Programms wird einer Variable ein Intervall zugeordnet, in dem der Wert der Variable sicher liegt. Für Schleifen wird ein Fixpunkt für die Intervalle der einzelnen Variablen berechnet. Die Schleife wird solange iteriert, bis sich das Intervall nicht mehr ändert. In den meisten Fällen bekommt man mit diesen Mechanismen nur grobe Abschätzungen. Schleifen mit der Zuweisung $x := x + 1$ führen oft zu einem Intervall $[1, \infty)$. In [7] wurde eine effiziente Methode vorgestellt, die diese Abschätzungen durchführt. Hier wird nicht nur der Datenfluß, sondern auch der Kontrollfluß berücksichtigt. Im allgemeinen ist festzustellen, daß für die Berechnung der Zeitfunktion, die abstrakte Interpretation nicht geeignet ist[13]. Es muß aber zwischen der abstrakten Interpretation als Theorie und den unter der Theorie vorgestellten Beispielen unterschieden werden. Die symbolische Analyse ist eine abstrakte Interpretation, jedoch ist der homomorphe Raum gleich mächtig. Es ist

[13]Die abstrakte Interpretation ist eine Theorie, die versucht, Aussagen über Programme zu machen. Dabei wird der ursprüngliche Problemraum in einen anderen, meistens einfacheren transformiert und homomorphe Operationen zum ursprünglichen Problemraum definiert. Das Programm wird im neuen Problemraum ausgeführt bzw. ausgewertet, um Aussagen über das Programm treffen zu können.

W ... symb. unentscheidbare Schleife
B ... Bedingung, die die Schleife W sicher ausführen läßt
E ... Eingaberaum des Programms

Markiere alle Variablen von W
$T = \{(x_1, \ldots, x_n) | \, v_1 = x_1 \wedge \ldots v_n = x_n \wedge B\} \cap E$
$U = \{(x_{i_1}, \ldots, x_{i_m}) | \, (x_1, \ldots, x_n) \in T\}$
$i_1, \ldots, i_m$ sind die Indizes der markierten Var.
Ersetze die Schleife W durch das Ergebnis der Erzeugerfunktion Z(U)

Die Erzeugerfunktion:
$Z(X) =$

Wenn X nicht $\emptyset$, dann nehme ein Element y aus X mit $(y_1, \ldots, y_m)$

if $v_{i_1} = y_1$ and $\ldots v_{i_m} = y_m$ *then*

W

else

$E(X - y)$

anderfalls

null

Abbildung 3.16: Umwandlungsalgorithmus für symbolisch unentscheidbare Schleifen

im Sinne von [47] keine Abstrahierung, sondern eine Übersetzung in den funktionalen Raum. Weitere Arbeiten über das Thema abstrakte Interpretation findet man in [9, 8, 20].

3.5.3 Schleifenauflösung mit Rekursionen

Die Induktionsvariablen(vgl. [62, 27]) einer Schleife können mit Hilfe von rekursiv definierten Folgen beschrieben werden. Durch geeignete Lösungsalgorithmen ist es möglich, die Schleife durch Zuweisungen an die Induktionsvariable zu ersetzten.

Das Beispiel in Abbildung 3.17 besitzt eine Schleife, die die Variable a solange um s erhöht, bis a größer oder gleich e ist. Die Schleife kann durch eine Zuweisung substituiert werden.

```
if a < e then
   while a < e
      a:=a+s
else
   ...
```

wird ersetzt durch

```
if a < e then
   a:= a + s * ((e - a) div s)
else
   ...
```

Abbildung 3.17: Beispiel: Auflösung mit Rekursionen

In vielen Fällen ist die Bestimmung eines exakten symbolischen Ausdrucks für die Variablen nicht möglich. Es kann aber eine Abschätzung des Wertes gefunden werden. Ein symbolisches Intervall gibt den Wertebereich an, in dem der Wert der Variable sicher liegt.

Aufstellen der Rekursionsgleichung

Der Wert einer Induktionsvariable (vgl. [1]) in einer Schleife ist abhängig vom Schleifendurchgang. Wir können den Wert der Variable v des i-ten Rekursionsschritts als Funktion $v(i)$ schreiben. $v(0)$ ist der Wert, den die Variable vor Ausführung der Schleife hat. Jeder weitere Schritt wird durch die Rekursionsgleichung $v(i+1) = \Xi(v(i)), \quad i \geq 0$ bestimmt. Wenn mehrere abhängige In-

duktionsvariablen in der Schleife vorkommen, ist die Rekursion ein rekursives Gleichungssystem.

Für das Aufstellen des rekursiven Gleichungssystems wird der Schleifenrumpf einmal symbolisch evaluiert. Vor der Auswertung des Schleifenrumpfs werden den Induktionsvariablen $v_j(i)$ zugeordnet, und sie erhalten somit den Wert des letzten Durchgangs am Ende des Schleifenrumpfs. Nach der Evaluierung des Schleifenrumpfs enthalten die einzelnen Variablen die symbolischen Werte für $v(i+1)$.

```
decl a,b,e,x,y:[1..100]

a:=x;
b:=y;
while a < e do
    a:=a+x;
    b:=2 * b
```

Abbildung 3.18: Beispielprogramm

Beispiel. Das Beispiel in Abbildung 3.18 enthält eine Schleife, deren DEF-Menge die Variablen a und b enthält[14]. Die Werte der Variablen vor der Schleife sind $\underline{x}$ und $\underline{y}$.

$$a(0) = \underline{x}$$
$$b(0) = \underline{y}$$

Im nächsten Schritt wird am Anfang des Schleifenrumpfs, den Variablen a und b, die Rekursionvariable $a(i)$ und $b(i)$ zugeordnet. Am Ende der Auswertung enthält die Schleife im Kontext für die zwei Variablen zwei symbolische Ausdrücke.

$$\ldots, (a, a(i) + \underline{x}), (b, 2 * b(i)), \ldots$$

[14]Variablen in der DEF-Menge einer Schleife sind potentielle Induktionsvariablen.

Sie entsprechen den Werten der Variablen am Ende des momentanen Durchgangs. Wir erhalten ein rekursives Gleichungssystem mit zwei Variablen. Beide Gleichungen sind voneinander unabhängig und können getrennt gelöst werden.

$$\begin{aligned} a(0) &= \underline{x} \\ a(i+1) &= a(i) + \underline{x}, \quad \text{wenn } i \geq 0 \end{aligned}$$

$$\begin{aligned} b(0) &= \underline{y} \\ b(i+1) &= 2 * b(i), \quad \text{wenn } i \geq 0 \end{aligned}$$

Mit Hilfe von mathematischen Verfahren kann das System gelöst werden.

$$\begin{aligned} a(i) &= (i+1) * \underline{x} \\ b(i) &= \underline{y} * 2^i \end{aligned}$$

Lösen von unbedingten Rekursionen

Rekursionen können nicht immer aufgelöst werden. If-Anweisungen erzeugen bedingte rekursive Gleichungssysteme, die in den wenigsten Fällen mit Hilfe von mathematischen Verfahren berechnet werden können. Die folgenden Formalismen behandeln nur unbedingte Rekursionen.

Summationsfaktoren. Dieses Verfahren (vgl. [29]) geht davon aus, daß die Rekursion erster Ordnung und linear ist und nur aus einer Rekursionsvariable besteht.

$$a(n)x_n = b(n)x_{n-1} + c(n), \quad n \geq 1 \tag{3.121}$$

Die Rekursion kann mittels Summation vereinfacht werden.

$$F(n) = \frac{\prod_{i=1}^{n-1} a(i)}{\prod_{j=1}^{n} b(j)} \tag{3.122}$$

Beide Seiten der Rekursion werden mit $F(n)$ multipliziert.

$$y_n = y_{n-1} + F(n)c(n) \tag{3.123}$$

wobei die Rekursion $y_n = b(n+1)F(n+1)x_n$ ist. Die letzte Rekursion erlaubt uns die ursprüngliche Rekursion als Summe anzugeben.

$$x_n = \frac{x_0 + \sum_{i=1}^{n} F(i)c(i)}{b(n+1)F(n+1)} \tag{3.124}$$

Erzeugende Funktionen. Das Verfahren (vgl. [29]) der erzeugenden Funktion ist mächtig, aber nicht algorithmisierbar.

Es wird für jede Variable eine Potenzreihe konstruiert. Die Koeffizienten der Potenzreihe sind die $v_j(i)$.

$$V_j(z) = \sum_i v_j(i) z^i \tag{3.125}$$

Anstatt die Rekursion direkt zu lösen, wird die Rekursion über die formalen Potenzreihen berechnet. Wenn das rekursive Gleichungssystem linear ist, wird jedes Vorkommnis von $v_j(i)$ durch $V_j(z)$ ersetzt — $v_j(i+1)$ durch $\frac{V_j(z)-v_j(0)}{z}$ substituiert. Konstante Terme c werden durch $\frac{c}{1-z}$ ersetzt.

Wir erhalten ein Gleichungssystem mit den Unbekannten $V_j(z)$. Das Gleichungssystem wird gelöst und für jedes $V_j(z)$ existiert eine Funktion in z. Sie besitzt für ein beliebiges z den gleichen Wert wie die gesuchte Potenzreihe an der Stelle z.

Um auf die $v_j(i)$ zu kommen, muß die Reihenentwicklung der Potenzreihe gefunden werden. Jeder Koeffizient in der Potenzreihe entspricht dem $v_j(i)$ der Rekursion.

Für nichtlineare Rekursionen wird ähnlich vorgegangen (vgl. [29]).

Beispiel

$$\begin{aligned} a(0) &= 0 \\ a(i+1) &= 3a(i) + 1, \quad \text{wenn } i \geq 0 \end{aligned}$$

Zunächst wird die Rekursionsvariable durch ihre Potenzreihe ersetzt.

$$\frac{A(z) - a(0)}{z} = 3A(z) + \frac{1}{1-z}$$

Danach wird $A(z)$ bestimmt, und eine Partialbruchzerlegung durchgeführt.

$$\begin{aligned} A(z) &= \frac{z}{(1-3z)(1-z)} \\ &= \frac{1}{2(1-3z)} + \frac{1}{2(1-z)} \\ &= \left[\frac{1}{2}\sum_i 3^i z^i\right] + \left[\frac{1}{2}\sum_i z^i\right] \end{aligned}$$

Die Koeffizienten der Reihe entsprechen den Werten der Variablen $a(i)$.

$$a(i) = \frac{(3^n - 1)}{2}, \quad \text{wenn } i \geq 0 \tag{3.126}$$

Symbolische Interpolation Rekursionen können mit Hilfe der Interpolation (vgl. [31],[36]) gelöst werden. Diese Methode ist elegant, hat aber den Nachteil, daß nur jene Rekursionen gelöst werden können, für die es ein k gibt, ab dem alle Differenzfolgen j-ter Ordnung, mit $j > k$, Nullfolgen sind.

Der Vorwärtsdifferenzoperator $\triangle$ erzeugt aus einer Folge eine neue Folge, deren Glieder aus der Differenz der urspünglichen Folge berechnet werden.

$$\triangle v(i) = v(i+1) - v(i) \tag{3.127}$$

$$\triangle^j v(i) = \triangle\left(\triangle^{j-1}\right), \quad j > 0 \tag{3.128}$$

$$\triangle^0 v(i) = v(i) \tag{3.129}$$

Die Rekursion für $v(i)$ kann nur dann mit Hilfe der symbolischen Interpolation aufgelöst werden, wenn das symbolische Interpolationskriterium erfüllt ist.

$$\exists k : \forall j > k : \triangle^j v(i) = [0] \tag{3.130}$$

Es wurde gezeigt, daß das Kriterium erfüllt ist, wenn die Lösung der Rekursion ein Polynom in i ist.

Die symbolische Interpolation wurde von der diskreten Newton-Interpolation abgeleitet. Für die Rekursion soll eine Funktion entwickelt werden, die den i-ten Wert der Rekursion direkt angibt. Die Funktion kann durch Interpolation berechnet werden, wenn das Interpolationskriterium hält. Nehmen wir an, daß für alle Punkte von $v(i)$ der Wert bekannt ist. Aufbauend auf den Werten wird die Newton Interpolation für $i = 1$ durchgeführt[15]. Wenn das Interpolationskriterium gilt, wird aus der Reihe eine endliche Summe. Die Differenzfolgen höherer Ordnung als k sind Nullfolgen und tragen nichts zum Ergebnis bei. Zur Berechnung der Funktion müssen höchstens die ersten $k + 2$ Werte der Rekursion bestimmt werden[16].

$$v(i) = \sum_{j=0}^{\infty} \triangle^j v(1) \binom{i-1}{j} \tag{3.131}$$

$$= \sum_{j=0}^{k} \triangle^j v(1) \binom{i-1}{j} \tag{3.132}$$

[15]Diese Vorgehensweise ist zunächst sinnlos: Wenn alle Punkte der Funktion $v(i)$ gegeben sind, ist die Interpolation unnötig.

[16]Schleifen müssen daher $k + 2$ Mal symbolisch exekutiert werden, um auf die ersten $k + 2$ Werte der Rekursion zu kommen.

Beispiel

$$a_i = a_{i-1} + 1, \quad \text{wenn } i \geq 0$$
$$b_i = b_{i-1} + a_{i-1}, \quad \text{wenn } i \geq 0$$

k wird bestimmt, indem schrittweise, rekursiv, die einzelnen Glieder der Rekursion und der Differenzfolgen berechnet werden.

i	a_i	b_i	$\triangle a_i$	$\triangle b_i$	$\triangle^2 b_i$
1	$a_0 + 1$	$b_0 + a_0$	1	$a_0 + 1$	1
2	$a_0 + 2$	$b_0 + 2a_0 + 1$	$\vdots$	$a_0 + 2$	$\vdots$
3	$\vdots$	$b_0 + 3a_0 + 3$	$\vdots$	$\vdots$	$\vdots$
4	$\vdots$	$\vdots$	$\vdots$	$\vdots$	

k ist für die Rekursion a_i gleich 1, da die Differenzfolge der zweiten Ordnung und alle höheren nur noch Nullfolgen sind. a_i wird daher mit der Formel 3.132 entwickelt.

$$\begin{aligned} a_i &= \sum_{j=0}^{k} \triangle^j a_1 \binom{i-1}{j} \\ &= (a_0 + 1)\binom{i-1}{0} + \binom{i-1}{1} \\ &= a_0 + i \end{aligned}$$

Für die Rekursion b_i ist $k = 2$. b_i ist daher

$$\begin{aligned} b_i &= \sum_{j=0}^{k} \triangle^j b_1 \binom{i-1}{j} \\ &= (b_0 + a_0)\binom{i-1}{0} + (a_0 + 1)\binom{i-1}{1} + \binom{i-1}{2} \\ &= \frac{i^2 + (2a_0 - 1)i + 2b_0}{2} \end{aligned}$$

Weitere Verfahren. Es gibt noch weitere Verfahren, die mit hypergeometrischen Reihen (vgl. [29]) arbeiten und daher noch eine größere Klasse von Rekursionen auflösen können. Weiters kann die Rekursion mit Hilfe von Differenzengleichungen berechnet werden.

In der Regel wird die symbolische Interpolation eine der effektivsten Mechanismen sein. Sie eignet sich gut für die symbolische Analyse, da der Aufwand, die Methode zu implementieren, relativ gering ist. Im Vergleich dazu brauchen alle anderen Methoden, massive Algebrawerkzeuge, deren Berechnungszeit in Abhängigkeit

der Datenmenge superexponentiell anwächst. Das Beispiel $a_i = 3a_{i-1} + 1$ kann mit der symbolischen Interpolation nicht gelöst werden, da das Interpolationskriterium nicht für diese rekursiv definierte Folge erfüllt ist. Die Klasse ist daher sehr eingeschränkt.

Abbruchbedingung

Die Abbruchbedingung gibt an, ob die Schleife nochmals durchlaufen wird, oder mit der nächsten Anweisung fortgefahren werden soll. In der symbolischen Analyse kann die evaluierte Abbruchbedingung der Schleife komplex sein. Einerseits ist sie aus Induktionsvariablen aufgebaut[17] und andererseits besteht sie aus Variablen, deren Werte symbolische Ausdrücke sind und in der Schleife konstant bleiben.

Die Schleife selbst kann durch eine Menge von Zuweisungen ersetzt werden, falls alle Rekursionen aufgelöst und der Abbruchindex gefunden wird. Der Abbruchindex ist die Iterationsstufe, nach der keine weitere Iteration der Schleife folgt.

Die Bedingung wird mit den $v_j(i)$ der Induktionsvariablen ausgewertet. D.h., daß vor einer Iteration die Schleifenbedingung überprüft wird[18].

Die evaluierte Bedingung ist ein Term in $\mathbb{F}_C$, der aus Relationen und logischen Operatoren besteht. Für jede Relation wird der Schleifenindex i auf die rechte Seite gebracht und die dabei enstehenden Fallunterscheidungen mit der Oder-Operation verknüpft. Relationen, die kein i enthalten, werden nicht verändert.

Beispiel. Die Relation in Abbildung 3.19 wird umgeformt. Die Umformung erzeugt aus der ursprünglichen Relation drei neue. Die Bedingungen der Fallunterscheidung werden mit der Und-Operation verknüpft. Das Ergebnis sind drei mit der Oder-Operation zusammengefaßte Terme. i befindet sich nur mehr auf der linken Seite der Relationen.

Im nächsten Schritt werden alle im Term vorkommenden Relationen in Kleiner-

[17] Enthält die Abbruchbedingung keine Induktionsvariablen, so kann die Bedingung nur in Abhängigkeit von anderen Variablen *true* bzw. *false* sein. Wenn es eine Variablenbelegung für diese Variablen gibt, unter der die Bedingung *true* ist, kann die Schleife nicht terminieren. Sie ist daher eine potentielle Endlosschleife.

[18] Würden wir die $v_j(i+1)$ Glieder nehmen, würde die Bedingung am Ende des Schleifenrumpfes getestet.

$$
\begin{aligned}
(ia < ci + 1) &= (ia - ci < 1) \\
&= (i(a-c) < 1) \\
&= \begin{cases} i < \frac{1}{a-c}, & \text{wenn } a - c > 0 \\ i > \frac{1}{a-c}, & \text{wenn } a - c < 0 \\ true, & \text{wenn } a = c \end{cases} \\
&= \left[i < \frac{1}{a-c} \text{ and } a - c > 0 \right] \text{ or} \\
&\quad \left[i > \frac{1}{a-c} \text{ and } a - c < 0 \right] \text{ or} \\
&\quad [true \text{ and } a = c] \\
&= \left[i < \frac{1}{a-c} \text{ and } a - c > 0 \right] \text{ or} \\
&\quad \left[i > \frac{1}{a-c} \text{ and } a - c < 0 \right] \text{ or} \\
&\quad [a = b]
\end{aligned}
$$

Abbildung 3.19: Beispiel: Umformung einer Relation

bzw. Größer-Relationen umgewandelt.

$$a = b \mapsto a \leq b \text{ and } a \geq b \tag{3.133}$$

$$a \neq b \mapsto a < b \text{ or } a > b \tag{3.134}$$

$$a \leq b \mapsto a < b + 1 \tag{3.135}$$

$$a \geq b \mapsto a > b - 1 \tag{3.136}$$

Es werden solange die i-Relationen umgeformt, bis nur $<$ bzw. $>$ Relationen vorhanden sind. Relationen, die den Index i nicht enthalten, bleiben unverändert.

Um den Abbruchindex symbolisch zu bestimmen, wird der Term in disjunktive Normalform(DNF) transformiert.

$$
\begin{aligned}
DNF(x) = &[t_{11} \text{ and } t_{12} \ldots \text{and } t_{1g_1}] \text{ or} \\
&[t_{21} \text{ and } t_{22} \ldots \text{and } t_{1g_2}] \text{ or} \\
&[t_{k1} \text{ and } t_{k2} \ldots \text{and } t_{kg_f}]
\end{aligned} \tag{3.137}
$$

Die Grundterme der DNF sind Relationen. Negierte Relationen werden mit dem Gesetz der „inversen Relation" in eine einfache Relation umgewandelt und an-

schließend wieder auf eine Kleiner- bzw. Größer-Relation gebracht, falls i in der Relation enthalten ist.

Für jeden Minterm[19] wird der Abbruchindex i getrennt bestimmt. In den Mintermen gibt es zwei Arten von Grundtermen: jene, die i-enthalten und jene, die von i unabhängig sind. Die von i abhängigen sind nur noch Kleiner- und Größerrelationen.

Die Größerrelationen geben für den Index eine untere Schranke an. Da aber die größte untere Schranke 0 ist, müssen alle anderen unteren Schranken kleiner gleich 0 sein. D.h., daß alle Relationen der Form $i > E$ in $E \leq 0$ umgeformt werden.

Alle Kleiner-Relationen sind obere Schranken für den Index. Die kleinste obere Schranke ist der Abbruchindex. Das Minimum der rechten Seiten aller Relationen der Form $i < E$ ist der gesuchte Abbruchindex. Existiert keine Kleiner-Relation im Minterm, so kann für diesen Fall kein Abbruchindex bestimmt werden.

Wenn ein Minterm die Form

$$
\begin{aligned}
m_j = {} & [(i > U_1)\,\text{and}\ldots(i > U_p)]\,\text{and} \\
& [(i < O_1)\,\text{and}\ldots(i < O_q)]\,\text{and} \\
& [t_1\,\text{and}\ldots\text{and}\,t_r]
\end{aligned}
\tag{3.138}
$$

hat, und t_1 bis t_r Relationen sind, die nicht den Index i enthalten, dann ist die symbolische Abbruchbedingung des Minterms

$$
\begin{aligned}
i = min(O_1, \ldots, O_q), \quad & \text{wenn}\,[t_1\,\text{and}\ldots\text{and}\,t_r]\,\text{and} \\
& [(U_1 \leq 0)\,\text{and}\ldots\text{and}(U_p \leq 0)]
\end{aligned}
\tag{3.139}
$$

Die symbolisch bestimmten Abbruchindizes der Minterme werden zu einer Fallunterscheidung zuammengefaßt. Wenn die min-Funktion symbolisch nicht aufgelöst werden kann, wird sie in die entsprechenden Fälle umgewandelt und in das Ergebnis eingesetzt.

$$
min(u_1, \ldots, u_t) = \begin{cases} u_1, & u_1 \leq u_2\,\text{and}\ldots\text{and}\,u_1 \leq u_t \\ u_2, & u_2 \leq u_1\,\text{and}\ldots\text{and}\,u_2 \leq u_t \\ \vdots & \\ u_t, & u_t \leq u_1\,\text{and}\ldots\text{and}\,u_t \leq u_t \end{cases}
\tag{3.140}
$$

[19]Eine Zeile in der DNF

Beispiel. Das Beispiel in Abbildung 3.20 besteht aus einer Schleife, die durch eine Zuweisung für die Induktionsvariable x ersetzt werden soll. Diese Transformation ist nur dann möglich, wenn die Rekursion von x gelöst werden kann.

$$x(0) = \underline{a} + \underline{b}$$
$$x(i) = x(0) - i$$

Im nächsten Schritt wird die Schleifenbedingung symbolisch ausgewertet. Für

```
decl a,b:[1..100];
     x,y:[-99,200]
x:=a+b;
if a < b then
   y := a - b
else
   y := b - a;

while x <> y do
   x := x - 1
```

Abbildung 3.20: Beispiel: Bestimmung des Abbruchindex

die Variable x wird $x(i)$ eingesetzt. Für die Variable y gibt es zwei verschieden symbolische Ausdrücke.

$$y = \begin{cases} \underline{a} - \underline{b}, & a < b \\ \underline{b} - \underline{a}, & a \geq b \end{cases}$$

Die Bedingung wird mit zwei Instanzen evaluiert. Das Ergebnis sind zwei mit der Oder-Operation verknüpfte Terme. Jeder Term enthält die mit der jeweiligen Instanz ausgewerteten Schleifenbedingung und der ererbten Bedingung der Instanz.

$$C = [(x(i) \neq \underline{a} - \underline{b}) \,\text{and}(a < b)] \,\text{or}[(x(i) \neq \underline{b} - \underline{a}) \,\text{and}(a \geq b)]$$

Die Rekursion für $x(i)$ wird in die Bedingung C eingesetzt.

$$C = [(\underline{a} + \underline{b} - i \neq \underline{a} - \underline{b}) \,\text{and}(a < b)] \,\text{or}$$
$$[(\underline{a} + \underline{b} - i \neq \underline{b} - \underline{a}) \,\text{and}(a \geq b)]$$

Danach wird der Index i für die Relationen explizit gemacht.

$$\begin{aligned} C = &[(i \neq 2\underline{b}) \,\text{and}(a < b)] \,\text{or} \\ &[(i \neq 2\underline{a}) \,\text{and}(a \geq b)] \end{aligned}$$

Im nächsten Schritt werden die Relationen, die i enthalten, auf $<$ bzw. $>$Relationen umgeformt.

$$\begin{aligned} C = &[\{(i < 2\underline{b}) \,\text{or}(i > 2\underline{b})\} \,\text{and}(a < b)] \,\text{or} \\ &[\{(i < 2\underline{a}) \,\text{or}(i > 2\underline{a})\} \,\text{and}(a \geq b)] \end{aligned}$$

Jetzt kann C in disjunktive Normalform gebracht werden. Schrittweise wenden wir das Distributivgesetz an und wir erhalten eine DNF(C) mit vier Mintermen.

$$\begin{aligned} C = &[(i < 2\underline{b}) \,\text{and}(a < b)] \,\text{or} \\ &[(i > 2\underline{b}) \,\text{and}(a < b)] \,\text{or} \\ &[(i < 2\underline{a}) \,\text{and}(a \geq b)] \,\text{or} \\ &[(i > 2\underline{a}) \,\text{and}(a \geq b)] \end{aligned}$$

Für jeden Minterm wird symbolisch der Abbruchindex bestimmt.

$$C = m_1 \,\text{or}\, m_2 \,\text{or}\, m_3 \,\text{or}\, m_4$$

Für m_1 ist $i < 2\underline{b}$ eine obere Schranke. D.h.,

$$i = 2\underline{b}, \quad \text{wenn}\, a < b$$

In m_2 gibt es keine obere Schranke. Das gleiche gilt für m_4. Nur für m_3 muß noch der Index bestimmt werden.

$$i = 2\underline{a}, \quad \text{wenn}\, a \geq b$$

Beide Indexberechnungen von m_1 und m_3 zusammengefaßt, ergeben folgende Fallunterscheidung.

$$i = \begin{cases} 2\underline{b}, & a < b \\ 2\underline{a}, & a \geq b \end{cases}$$

Nachdem der Abbruchindex bestimmt worden ist, können wir den Wert der Induktionsvariable x nach der Schleife direkt angeben.

$$\begin{aligned} x &= \underline{a} + \underline{b} - i \\ &= \underline{a} + \underline{b} - \begin{cases} 2\underline{b}, & a < b \\ 2\underline{a}, & a \geq b \end{cases} \\ x &= \begin{cases} \underline{a} - \underline{b}, & a < b \\ \underline{b} - \underline{a}, & a \geq b \end{cases} \end{aligned}$$

Das Programm in Abbildung 3.21 ist semantisch mit dem Programm in Abbildung 3.20 äquivalent.

```
decl a,b,x,y:[1..100];

x:=a+b;
if a < b then
   y := a - b
else
   y := b - a;

if a < b then
   x := a - b
else
   x := b - a
```

Abbildung 3.21: Beispiel: Aufgelöste Schleife

Bedingte rekursive Gleichungssysteme

Induktionsvariablen, für die eine Zuweisung in einer If-Anweisung existiert, erzeugen ein rekursives bedingtes Gleichungssystem. Für diese Systeme gibt es keine allgemeine Lösungsmethode, obwohl sie in der Analyse von Programmen sehr oft vorkommen. Die symbolische Analyse ist für diese Klasse von Rekursionen sehr aufwendig. Ein exakter Algorithmus existiert nicht. Es gibt aber gültige Transformationen, die das System schrittweise vereinfachen.

Ein μ-Operator wird verwendet, um das bedingte rekursive Gleichungssystem darzustellen. Der Operator μ ist ein Operator in der Funktionenalgebra. Er

hat als Argumente die Startwerte der Induktionsvariablen, die Abbruchbedingung der Rekursion, die bedingten rekursiven Gleichungen und eine Induktionsvariable, deren Wert nach dem letzten Schritt der Rekursion[20] als Resultat des Operators genommen wird.

$$\mu\left(v_j, \{v_1(0) = \ldots, \ldots, v_\eta(0) = \ldots\}, C, \{v_1(i+1) = \ldots, \ldots, v_\eta(i+1) = \ldots\}\right)$$

Die Schleife hat η Induktionsvariablen. Das Resultat der μ-Operation ist die Induktionsvariable v_j, die im ersten Iterationsschritt für die symbolisch evaluierte Bedingung C *false* ergibt. Die Startwerte der Rekursion legt das zweite Argument fest. Das letzte Argument ist eine Menge von Rekursionsgleichungen.

In der symbolischen Analyse kann eine Schleife durch eine Menge von Zuweisungen für die Induktionsvariablen ersetzt werden. Jede Induktionsvariable der Schleife bekommt eine μ-Operation zugeordnet. Die Resultatsvariable der μ-Operation ist die Induktionsvariable selbst. Alle anderen Argumente des μ-Operators sind für die Induktionsvariablen ident.

$$\begin{array}{l} \textit{while} \ldots \\ \quad a = \ldots \\ \quad b = \ldots \end{array} \iff \begin{array}{l} a = \mu(a, \ldots) \\ b = \mu(b, \ldots) \end{array} \tag{3.141}$$

Beispiel. Das Beispiel in Abbildung 3.18 hat zwei Induktionsvariablen a und b. Die Schleife wird in Form eines Gleichungsystems dargestellt. Das Beispiel enthält keine Fallunterscheidung in der Rekursion.

$$\begin{aligned} a = \mu(a(i), \{a(0) = \underline{x}, b(0) = \underline{y}\}, \\ \quad a(i) < e, \{a(i+1) = a(i) + \underline{x}, b(i+1) = 2 * b(i)\}) \\ b = \mu(b(i), \{a(0) = \underline{x}, b(0) = \underline{y}\}, \\ \quad a(i) < e, \{a(i+1) = a(i) + \underline{x}, b(i+1) = 2 * b(i)\}) \end{aligned}$$

Vereinfachung und Auflösung

Die μ-Operation läßt eine Anzahl von Vereinfachungen zu, sodaß Rekursionen leichter aufgelöst werden können.

Manche Rekursionen sind degeneriert, sodaß für den Index 0, die Bedingung des μ-Operators *false* ist. Anstatt des μ-Operators, kann der Startwert der Induktionsvariable genommen werden. Ein erweiterter Test ist die Überprüfung, ob die

[20] wird durch die Abbruchbedingung bestimmt

$M = \{v_j\} \cup USE(C)$
$M_n = \emptyset$
solange $M \neq M_n$
    $M_n = M$
    für jedes $v_j(i+1) = expr$
        $M_n = M_n \cup USE(expr)$
$M_n = M$

Abbildung 3.22: Markierungsalgorithmus für den μ-Operator

Bedingung, die die μ-Operation ausführen läßt, mit der Abbruchbedingung der μ-Operation *false* ist.

$$\mu\left(v_j, \{\ldots v_j(0) = expr\}, false, \{\ldots\}\right) = expr \tag{3.142}$$

Für alle nicht degenerierten Rekursionen werden jene Induktionsvariablen aus μ entfernt, die nicht für die Berechnung des Resultats benötigt werden. Ausgehend von der Resultatsvariable und den Induktionsvariablen in der Bedingung, werden alle für die Berechnung notwendigen Variablen gekennzeichnet. Der Algorithmus wird solange wiederholt, bis keine weiteren Induktionsvariablen zum Markieren gefunden werden. Alle nicht markierten Induktionsvariablen können aus dem μ-Operator genommen werden (siehe Abbildung 3.22).

Im vorigen Beispiel kann in der μ-Operation der Variable a, die Induktionsvariable $b(i)$ entfernt werden, da b weder für die Abbruchbedingung noch für die Berechnung von $a(i)$ gebraucht wird.

Nachdem alle unnötigen Induktionsvariablen aus μ entfernt worden sind, werden alle unbedingten Rekursionen aufgelöst. μ enthält nur noch bedingte Gleichungen.

Im nächsten Schritt versuchen wir alle statischen[21] Bedingungen bzw. Teile von Bedingungen, die statisch sind, aus der μ-Operation herauszuziehen. Die Fallunterscheidung der statischen Bedingung wird vor der μ-Operation durchgeführt. Für den Fall, daß die Bedingung *false* ist, wird eine μ-Operation benötigt und für

[21] unabhängig von den Induktionsvariablen der Schleife

den Fall, daß die Bedingung *true* ist, die zweite μ-Operation verwendet.

$$\mu\left(v_j, \{\ldots\}, C, \{\ldots, v_k(i+1) = \begin{cases} \ldots, & \ldots x \ldots \\ \vdots & \end{cases}\right) \Longleftrightarrow$$

$$\begin{cases} \mu\left(v_j, \{\ldots\}, C, \{\ldots, v_k(i+1) = \begin{cases} \ldots, & \ldots true \ldots \\ \vdots & \end{cases}\right) & , x \\ \mu\left(v_j, \{\ldots\}, C, \{\ldots, v_k(i+1) = \begin{cases} \ldots, & \ldots false \ldots \\ \vdots & \end{cases}\right) & , \text{not } x \end{cases}$$

Alle statischen Bedingungen bzw. Teilbedingungen einer μ-Operation werden herausgezogen, sodaß nur noch Bedingungen abhängig von den Induktionsvariablen vorhanden sind.

Das folgende Beispiel enthält zwei statische Bedingung, die schrittweise herausgezogen werden.

$$\mu(x, x(0) = 10, x(i) < 100, x(i+1) = \begin{cases} x(i)+1, & a < b \\ x(i)+2, & a \geq b \end{cases})$$

$$= \begin{cases} \mu(x, x(0) = 10, x(i) < 100, x(i+1) = \begin{cases} x(i)+1, & true \\ x(i)+2, & a \geq b \end{cases}), & a < b \\ \mu(x, x(0) = 10, x(i) < 100, x(i+1) = \begin{cases} x(i)+1, & false \\ x(i)+2, & a \geq b \end{cases}), & \text{not}(a < b) \end{cases}$$

$$= \begin{cases} \mu(x, x(0) = 10, x(i) < 100, x(i)+1), & a < b \\ \mu(x, x(0) = 10, x(i) < 100, x(i)+2), & a \geq b \end{cases}$$

3.5.4 Abschätzungen

In den meisten Fällen können bedingte Rekursionen nicht aufgelöst werden. Es ist jedoch möglich, eine symbolische Abschätzung für Rekursionen zu finden. In [5] wird eine Abschätzung für Rekursionen innerhalb eines symbolischen Intervalls angegeben. Die Schleifenbedingungen müssen dazu eine bestimmte Struktur aufweisen.

Abschätzungen können leicht gefunden werden. Die Präzision der symbolischen Analyse hängt aber von der Qualität einer Abschätzung ab. Es ist daher sinnvoll, einen genauen Plan für die Analyse von Schleifen einzuhalten. Zunächst wird versucht, alles exakt zu lösen. Wenn die exakte Analyse die μ-Operatoren

nicht weiter auflösen kann, dann wird eine geeignete Abschätzung gesucht und angewendet. Danach wird wieder mit der exakten Analyse fortgefahren.

3.6 Prozeduren

Prozeduren (vgl. [32]) sind eine Erweiterung des bisherigen Programmodells. Jede Prozedur wird getrennt analysiert und in eine funktionale Definition transformiert. Die getrennte Analyse setzt aber voraus, daß es keine globalen Variablen im Programmsystem gibt. Seiteneffekte können zunächst nicht behandelt werden.

Die Prozedur wird in der symbolischen Analyse in eine Menge von Funktionen transformiert. Die Argumente der Funktionen sind alle **in** und **inout** Variablen $(v_1, \ldots, v_p)$. Für alle **out** und **inout** Variablen $(o_1, \ldots, o_q)$ existiert eine Funktion über den Parametern.

(3.143)

proc $X(v_1, \ldots, v_p)$
...
endproc

$$\Longleftrightarrow X(v_1, \ldots, v_p) = \Big(X_{o_1}(v_1, \ldots, v_p), \ldots, X_{o_q}(v_1, \ldots, v_p)\Big)$$

Vier unterschiedliche Arten von Variablen können in einer Prozedur verwendet werden:

1. **local.** Lokale Variaben sind Variablen, die zu Beginn undefiniert sind. Jede Instanz eines Aufrufs erhält einen eigenen Satz von lokalen Variablen. In der symbolischen Analyse werden sie wie im bisherigen Ansatz behandelt. Zu Beginn wird ihnen das Symbol $\perp$ zugeordnet, das den undefinierten Zustand kennzeichnet. Die für die Variable erzeugte Funktion wird am Ende der Analyse nicht mehr weiter berücksichtigt. Sie wird nur während der Analyse als Zwischenergebnis verwendet.

2. **inout.** Zu Beginn der Analyse erhalten „Call by address" Variablen den symbolischen Wert des Parameters $\underline{v_i}$. Die erzeugte Funktion wird als Ergebnis o_j der Prozedur weiterverwendet.

3. **in.** „Call by value" Variablen erhalten zu Beginn der Analyse den Wert des Parameters. Am Ende der Analyse wird die Funktion der **in** Variable nicht mehr weiterverwendet.

4. **out.** Eine **out** Variable ist eine Variable, die zu Beginn der Analyse undefiniert ($\perp$) ist. Für die Variable wird das Ergebnis der Analyse im Ausgabevektor der Prozedur aufgenommen.

Der Aufruf einer Prozedur ist semantisch mit einer Menge von Zuweisungen äquivalent. Die Parametervariablen $(o_1, \ldots, o_q)$ der Prozedur weisen den übergebenen Variablen neue Werte zu. Der neue Wert der für die Parametervariable o_i übergebenen Variable ist die Funktion $X_{o_i}(v_1, \ldots, v_p)$.

Wenn die Prozedur nicht rekursiv definiert wurde, kann direkt die Funktion eingesetzt werden, andernfalls darf nur der Funktionsname verwendet werden. Das Auflösen von Rekursionen wurde bereits im Unterkapitel über Schleifen behandelt. Jedoch sei hier nochmals angemerkt, daß rekursive Prozeduren (vgl. [6]) entscheidbar sind[22], aber kein allgemeiner Algorithmus angegeben werden kann, um sie aufzulösen.

Wenn keine rekursiven Prozeduren im Programmsystem vorhanden sind, kann jedes Programmsystem mit mehreren Prozeduren in ein Hauptprogramm ohne Prozeduren transformiert werden (vgl. ,,Inline Expansion" [1]).

Beispiel. Das Beispiel in Abbildung 3.23 deklariert zwei Prozeduren. Die Prozedur add hat als Argumente a, b und z. Die ersten zwei Argumente sind ,,Call by value" Parameter. Das letzte Argument ist ein **out** Parameter. Die Prozedur hat in der funktionalen Definition zwei Parameter und eine Ausgabekomponente.

Für die Analyse des Beispiels wird ein Kontext ohne Pfadbedingung verwendet. Das Durchrechnen mit den unterschiedlichen Methoden bleibt dem Leser überlassen.

$$\mathrm{add}(\underline{a}, \underline{b}) = (\mathrm{add}_z(\underline{a}, \underline{b}))$$

In der Analyse wird für die Prozedur add folgender Anfangskontext verwendet.

$$\mathrm{ctx}_0 = \{(a, \underline{a}), (b, \underline{b}), (t, \perp), (z, \perp)\}$$

Die Funktionen θ und θ^{-1} müssen die unterschiedlichen Arten von Parameter berücksichtigen.

[22] aufgrund des endlichen Eingaberaums

```
decl x,y:[1..100];

proc add(in a:[1..100]; in b:[1..100]; out z:[1..100])
local t:[1..100];

   t:=a + b;
   z:=t;
endproc

proc sub(in a:[1..100];inout b:[1..100])
   b:=a - b;
endproc

....

  add (x,y+10,x);
  sub (y,x);
```

Abbildung 3.23: Beispiel für Prozeduren in der symbolischen Analyse

Nach der symbolischen Evaluierung der Prozedur add wandelt die Funktion θ^{-1} den letzten Kontext der Auswertung

$$\mathrm{ctx}_f = \{(a, \underline{a}), (b, \underline{b}), (t, \underline{a} + \underline{b}), (z, \underline{a} + \underline{b})\}$$

in die Funktion

$$\begin{aligned} \mathrm{add}(\underline{a}, \underline{b}) &= (\mathrm{add}_z(\underline{a}, \underline{b})) \\ &= (\underline{a} + \underline{b}) \end{aligned}$$

um. Analog zur Prozedur add wird die Analyse für die Prozedur sub durchgeführt. Die Prozedur hat zwei Argumente. Das erste ist eine **in** Parametervariable und das zweite eine **inout** Parametervariable. Die funktionale Definition der Funktion

hat daher folgende Form:

$$\begin{aligned} \mathrm{sub}(\underline{a}, \underline{b}) &= (\mathrm{sub}_b(\underline{a}, \underline{b})) \\ &= \underline{a} - \underline{b} \end{aligned}$$

Das Hauptprogramm des Beispiels in Abbildung 3.23 besteht aus zwei Aufrufen. Der erste weist der Variable x einen neuen Wert zu[23]. Die Variable x erhält nach dem Aufruf den Wert:

$$\begin{aligned} \mathrm{ctx} &= \{\ldots, (x, \mathrm{add}_z(\underline{x}, \underline{y} + 10)), \ldots\} \\ &= \{\ldots, (x, \underline{x} + \underline{y} + 10), \ldots\} \end{aligned}$$

In der nächsten Anweisung wird die Prozedur sub aufgerufen. x erhält wieder einen neuen Wert.

$$\begin{aligned} \mathrm{ctx} &= \{\ldots, (x, \mathrm{sub}_z(\underline{y}, \underline{x} + \underline{y} + 10)), \ldots\} \\ &= \{\ldots, (x, -\underline{x} - 10), \ldots\} \end{aligned}$$

Globale Variablen

Um globale Variablen trotzdem verwenden zu können, werden sie im Hauptprogramm lokal deklariert und bei jedem Prozeduraufruf als ,,Call by address" Parameter mitgegeben.

Dieser naive Ansatz ist in der symbolischen Analyse nicht effizient, da viele globale Variaben in den Prozeduren nicht modifiziert und gelesen werden. Die wenigsten globalen Variablen werden in Prozeduren wirklich benötigt. Durch geeignete Algorithmen kann die Menge der globalen Variablen, die jeder Prozedur übergeben werden, auf eine Teilmenge der verwendeten reduziert werden. Wenn globale Variablen nur gelesen werden, können sie als ,,Call by value" Parameter definiert werden.

3.7 Arrays

Die symbolische Analyse von endlichen Arrays wird zunächst für den eindimensionalen Fall betrachtet. Der Typ "Array" wird durch die Anzahl der Array-Elemente

[23] x ist das 3. Argument des Prozeduraufrufs

k und durch den Wertebereich $[u, o]$ der Elemente bestimmt. Jedes Element des Arrays besitzt den selben Wertebereich.

$$\mathcal{A}_{(k,[u,o])} = \underbrace{[u,o] \times \cdots \times [u,o]}_{k}, \quad u \leq o \tag{3.144}$$

Die Funktion rg ermittelt für Arrays des Typs $\mathcal{A}_{(k,[u,o])}$ den Wertebereich der Array-Elemente.

$$\mathrm{rg}(x) = [u,o], \text{wenn } x \in \mathcal{A}_{(k,[u,o])} \tag{3.145}$$

Die Funktion dim liefert die Dimension eines Arrays in τ-Simple.

$$\dim(x) = k, \text{wenn } x \in \mathcal{A}_{(k,[u,o])} \tag{3.146}$$

In der Sprache τ-Simple wird über einen Index lesend bzw. schreibend auf einzelne Elemente eines Arrays zugegriffen. Für die Zuweisung wird in der Standardsemantik die Funktion vecset verwendet.

(3.147)
$$\mathrm{val}_S(vec[E_1] := E_2, \mathrm{env}) \mapsto \mathrm{vecset}(\mathrm{env}, vec, \mathrm{val}_E(E_1, \mathrm{env}), \mathrm{val}_E(E_2, \mathrm{env})),$$
$$\text{wenn } \mathrm{val}_E(E_2, \mathrm{env}) \in \mathrm{rg}(vec) \wedge 1 \leq \mathrm{val}_E(E_1, \mathrm{env}) \leq \dim(vec)$$

E_1 ist ein Audruck, der das Array-Element im Array vec indiziert. E_2 ist der neue Wert des E_1-ten Elements. Die Funktion val_S ist genau dann definiert, wenn der Index innerhalb der Dimension des Arrays und der neue Wert des Elements im Wertebereich liegt.

vecset wird analog zur Funktion vset für Skalare definiert.

$$\begin{aligned} \mathrm{vecset} : &\mathrm{ENV} \times \mathcal{A}_{(k,[u,o])} \times \mathbb{Z} \times \mathbb{Z} \to \mathrm{ENV} \\ &\mathrm{vecset}(\mathrm{env}, x, i, a) = \\ &\quad \{\ldots(x, (w_1, \ldots, w_{i-1}, a, w_{i+1}, \ldots, w_n))\ldots\} \end{aligned} \tag{3.148}$$

Vom Array x in der Variablenumgebung env wird das i-te Element mit dem Wert a überschrieben.

Der lesende Zugriff auf Arrays erfolgt über die Indizierung eines Elements. Die Standardsemantik verwendet die Funktion vecget.

(3.149)
$$\mathrm{val}_E(vec[E], \mathrm{env}) = \mathrm{vecget}(\mathrm{env}, vec, \mathrm{val}_E(E, \mathrm{env})), \quad \text{wenn } 1 \leq \mathrm{val}_E(E, \mathrm{env}) \leq \dim(vec)$$

Die Zugriffsfunktion vecget wird analog zu vget definiert.

(3.150)

$$\begin{aligned}\text{vecget} : &\text{ENV} \times \mathcal{A}_{(k,[u,o])} \times \mathbb{Z} \to \mathbb{Z} \\ &\text{vecget}(\{\ldots(x,(w_1,\ldots,w_i,\ldots,w_n))\ldots\},x,i) = \\ &w_i\end{aligned}$$

Die Funktionenalgebra $\mathbb{F}$ wird mit einem neuen symbolischen Datentyp für Arrays erweitert. Er erlaubt uns, ähnlich wie mit Skalaren, symbolische Ausdrücke für Arrays zu berechnen. Zwei Operatoren werden in der Array-Algebra verwendet. Der erste ist für das Lesen eines Elements, der zweite für das Schreiben einer Zelle im Array zuständig.

(3.151) $$\xi_r : \mathcal{A}_{(k,[u,o])} \times \mathbb{F}_E \to \mathbb{F}_E$$

(3.152) $$\xi_w : \mathcal{A}_{(k,[u,o])} \times \mathbb{F}_E \times \mathbb{F}_E \to \mathcal{A}_{(k,[u,o])}$$

Für beide Operatoren gelten folgende Rechenregeln.

(3.153)

$$\text{Leseregel} \quad \xi_r(\xi_w(x,i,w),j) = \begin{cases} w, & i = j \\ \xi_r(x,j), & sonst \end{cases}$$

(3.154)

$$\text{Kürzungsregel} \quad \xi_w(\ldots\xi_w(x,i,w)\ldots,j,v) = \begin{cases} \xi_w(\ldots x\ldots,j,v), & i = j \\ \xi_w(\ldots\xi_w(x,i,w)\ldots,j,v), & sonst \end{cases}$$

x ist Element aus $\mathcal{A}_{(k,[u,o])}$ und i,w,v,j sind Elemente aus $\mathbb{F}_E$.

Jede Array-Manipulation kann durch ξ_w-Ketten dargestellt werden, ohne daß die konkreten Werte des Arrays bekannt sind.

Beispiel. Das Programm in Abbildung 3.24 mit dem Array a schreibt zuerst in die erste Zelle den Wert $10 + \underline{y}$ und in das zweite Element den Wert $20 + \underline{z}$[†]. Die Variable a erhält in der symbolischen Evaluierung den Wert:

$$x = \xi_w(\xi_w(\underline{x}, 1, 10 + \underline{y}), 2, 20 + \underline{z})$$

[†] $\underline{y}$ und $\underline{z}$ sind Skalare.

```
decl a:array[1..2] of [1..100];
     y,z:[1..100];

a[1] := 10+y;
a[2] := 20+z
```

Abbildung 3.24: Beispiel für Arrays

3.7.1 Rekursiv definierte Arrays

Schleifen können in Abhängigkeit der Eingabe sehr viele ξ_w-Ketten erzeugen. Die Anzahl der Array-Ketten, die eine Schleife produzieren kann, ist endlich[24], jedoch ist die Aufzählung aller möglichen Ketten praktisch nicht durchführbar. Wir verwenden daher eine rekursive Beschreibung, um Arrays, die sich in einer Schleife ändern, darzustellen.

$$a(0) = ... \tag{3.155}$$
$$a(n+1) = \Xi(a(n)), n \geq 0$$

$a(n)$ ist ein Array vom Typ $\mathcal{A}_{k,[u,o]}$ und Ξ eine beliebige Funktion, die das Array des vorhergehenden Schleifendurchgangs verwendet. $a(n)$ ist ein Array und *nicht* das n-te Element des Arrays a.

Beispiel. Das Beispiel Abbildung 3.25 beschreibt das Array a mit den Werten von $m-1$ bis 1 in absteigender Folge. Danach werden die im Array a gespeicherten Werte als Index für w verwendet.

Die symbolische Analyse der ersten Schleife des Programms erzeugt ein rekursives Gleichungssystem.

$$\begin{aligned} i(0) &= 1 \\ i(n+1) &= i(n) + 1, \quad n \geq 0 \\ a(0) &= \underline{a} \\ a(n+1) &= \xi_w(a(n), i(n), m - i(n)), \quad n \geq 0 \end{aligned}$$

[24]Der Eingaberaum ist endlich.

```
decl i,m:[1..100];
     a,w:array [1..100] of [1..100];

for i:=1 to m-1 do
   a[i] := m - i
for j:=1 to m-1 do
   w[a[i]]=j+1
```

Abbildung 3.25: Beispiel für Arrays

Die Rekursion für $i(n+1)$ kann zur Gänze aufgelöst werden. Die Variable i ist im Gegensatz zu a[†] ein Skalar und es gibt geeignete Algorithmen zur Auflösung. Das Array a kann nur rekursiv definiert werden. Eine andere Alternative ist die Bestimmung aller ξ_w-Ketten (siehe „Exekution von Schleifen").

Da $i(n+1)$ vollständig aufgelöst werden kann, setzen wir das Ergebnis von $i(n)$ in die Definition von $a(n+1)$ ein. Eine einfachere Form der Rekursion entsteht.

$$\begin{aligned} i(n) &= n+1, \quad n \geq 0 \\ a(0) &= \underline{a} \\ a(n+1) &= \xi_w(a(n), n+1, m-n-1), \quad n \geq 0 \end{aligned}$$

Das Array a kann nicht weiter vereinfacht werden. Die Kürzungsregel ist nicht anwendbar. Der Index des ξ_w-Operators ist eine streng monoton steigende Funktion und es gibt daher in der rekursiv definierten ξ_w-Kette keine zwei Indizes, die gleich sind. Die Schleife wird mit dem Index $m-1$ abgebrochen. D.h., der Wert von a nach der ersten Schleife ist

$$a = \begin{cases} a(m-1), & m \geq 2 \\ \underline{a}, & sonst \end{cases}$$

In der zweiten Schleife wird folgendes rekursives Gleichungssystem aufgestellt:

$$\begin{aligned} j(0) &= 1 \\ j(n+1) &= j(n)+1, \quad n \geq 0 \\ w(0) &= \underline{w} \\ w(n+1) &= \xi_w(w(n), \xi_r(a(m), j(n)), j(n)+1), \quad n \geq 0 \end{aligned}$$

[†]In der Rekursion $a(n+1)$ wird nicht das n+1-te Element definiert, sondern ein neuer Array, der von seinem Vorgänger a(n) abgeleitet wird.

Die Rekursion $j(n)$ ist eine skalare Funktion und kann wieder aufgelöst und in die Rekursion für das Array w eingesetzt werden.

$$\begin{aligned} j(n) &= n+1, \quad n \geq 0 \\ w(0) &= \underline{w} \\ w(n+1) &= \xi_w(w(n), \xi_r(a(m), n+1), n+2), \quad n \geq 0 \end{aligned}$$

Im nächsten Schritt wird der Index der ξ_w-Operation berechnet. Da der Index der rekursiven Definition von a eine streng monoton wachsende Folge ist, kann die Leseregel rekursiv angewendet werden, um den Wert des Elements mit dem Index $n+1$ zu erhalten.

$$\begin{aligned} \xi_r(a(m), n+1) &= \begin{cases} m-n-1, & n+1 \geq m \wedge n \geq 0 \\ \xi_r(\underline{a}, n+1), & sonst \end{cases} \\ &= \begin{cases} m-n-1, & m \leq 1 \wedge n \geq 0 \\ \xi_r(\underline{a}, n+1), & sonst \end{cases} \end{aligned}$$

Der vereinfachte Index wird in die Definition von w eingesetzt.

$$\begin{aligned} w(0) &= \underline{w} \\ w(n+1) &= \begin{cases} \xi_w(w(n), m-n-1, n+2), & n \geq 0 \wedge m \geq 2 \\ \xi_w(w(n), \xi_r(\underline{a}, n+1), n+2), & n \geq 0 \wedge m < 2 \end{cases} \end{aligned}$$

Der Wert von w ist nach der zweiten Schleife $w(m-1)$, falls m größer als 2 ist. Andernfalls ist er $\underline{w}$.

$$w = \begin{cases} w(m-1), & m \geq 2 \\ \underline{w}, & sonst \end{cases}$$

3.7.2 Mehrdimensionale Arrays

Linearisierung

Mehrdimensionale Arrays lassen sich mit Hilfe des Horner-Schemas (vgl. [60]) auf eindimensionale Arrays abbilden. Ein mehrdimensionaler Array a mit dem Typ

$$\mathcal{A}_{(k_1,[u,o])} \times \mathcal{A}_{(k_2,[u,o])} \cdots \times \mathcal{A}_{(k_n,[u,o])} \tag{3.156}$$

hat n Indizes und kann mit Hilfe einer geeigneten Abbildungsfunktion λ auf einen eindimensionalen Array a' reduziert werden.

(3.157) $$a'[\lambda(i_1, \ldots, i_n)] = a[i_1, \ldots, i_n]$$

In einem Programm werden alle mehrdimensionalen Arrays auf eindimensionale umgewandelt und mit den ξ_w-Ketten analysiert (siehe Abbildung 3.26).

Der Nachteil der Linearisierung sind komplizierte Index-Ausdrücke, die symbolisch schwer behandelbar sind. Eine explizite Behandlung der Indizes eines mehrdimensionalen Arrays kann in vielen Fällen einfacher durchgeführt werden.

```
decl a:array [1..2,1..2] of [1..100];
     i,j:[1..100];

for i:=1 to 2 do
   for j:=1 to 2 do
      a[i,j]:=0
```

wird zu

```
decl a´:array [1..4] of [1..100];
     i,j:[1..100];

for i:=1 to 2 do
   for j:=1 to 2 do
      a´[2*i+j-2]:=0
```

Abbildung 3.26: Umwandlung eines mehrdimensionalen Arrays

Explizite Behandlung von mehrdimensionalen Arrays

Mehrdimensionale Arrays können durch die Bildung einer Funktionenklasse für die Operationen ξ_w und ξ_r behandelt werden.

(3.158)
$$\xi_r : \mathcal{A}_{(k_1,[u,o])} \times \ldots \mathcal{A}_{(k_n,[u,o])} \times \mathbb{F}_E \rightarrow \mathbb{F}_E$$

(3.159)
$$\xi_w : \mathcal{A}_{(k_1,[u,o])} \times \ldots \mathcal{A}_{(k_n,[u,o])} \times \mathbb{F}_E \times \mathbb{F}_E \rightarrow \mathcal{A}_{(k_1,[u,o])} \times \ldots \mathcal{A}_{(k_n,[u,o])}$$

Die Operation ξ_r hat als Argument ein n-dimensionales Array mit n Indizes. Die Operation ξ_w verändert das Element, das über die n-Indizes eines n-dimensionalen Arrays bestimmt wird.

Die Gesetze für die Operationen werden analog zu den Regeln für eindimensionale Arrays definiert.

(3.160)

$$\text{Leseregel} \quad \xi_r(\xi_w(x, i_1, \ldots, i_n, w), j_1, \ldots, j_n) = \begin{cases} w, & i_1 = j_1 \wedge \ldots i_n = j_n \\ \xi_r(x, j_1, \ldots, j_n), & sonst \end{cases}$$

(3.161)

$$\text{Kürzungsregel} \quad \xi_w(\ldots \xi_w(x, i_1, \ldots, i_n, w) \ldots, j_1, \ldots, j_n, v) = \begin{cases} \xi_w(\ldots x \ldots, j_1, \ldots, j_n, v), & i_1 = j_1 \wedge \ldots i_n = j_n \\ \xi_w(\ldots \xi_w(x, i_1, \ldots, i_n, w) \ldots, j_1, \ldots, j_n, v), & sonst \end{cases}$$

```
decl a:array[1..2,1..2] of [1..100];
     x,y,z:[1..100];

a[1,1] := x;
a[2,1] := y;
z      := a[1,1]
```

Abbildung 3.27: Beispiel für explizite Behandlung von mehrdimensionalen Arrays

Beispiel. Im Programm Abbildung 3.27 wird das Element $(1, 1)$ des Arrays a auf den Wert x und das Element $(2, 1)$ auf den Wert y gesetzt.

$$a = \xi_w(\xi_w(\underline{a}, 1, 1, \underline{x}), 2, 1, \underline{y})$$

Die skalare Variable z erhält den Wert des Elements $(1,1)$. Schrittweise wird die Leseregel angewendet.

$$\begin{aligned} z &= \xi_r(a,(1,1)) \\ &= \xi_r(\xi_w(\xi_w(\underline{a},1,1,\underline{x}),2,1,\underline{y}),1,1) \\ &= \xi_r(\xi_w(\underline{a},1,1,\underline{x}),1,1) \\ &= \underline{x} \end{aligned}$$

3.8 Ein-/Ausgabe

Ein-/Ausgabe wird in Form von zwei unendlich lange Bändern dargestellt. Vom Eingabeband werden einzelne Symbole gelesen, auf das Ausgabeband werden Symbole geschrieben. Da beide Bänder unendlich lang sind, wird die Eigenschaft der Entscheidbarkeit zerstört. In Echtzeitprogrammen sind daher E/A-Operationen nicht zu verwenden. Eine darüberliegende Ein-/Ausgabeschicht führt das Ein-/Ausgabeprotokoll durch und muß mit speziell dafür entwickelten Methoden verifiziert werden (vgl. [40]).

Ein Programm mit Ein- /Ausgabe kann symbolisch analysiert werden, wenn die Read-/Write-Anweisung in Array-Operationen umgewandelt werden. Für jeden Datentyp existiert ein globales Array für das Lesen und ein globales Array für das Schreiben. Zusätzlich werden zwei Indexvariablen definiert, die für jede Ein- /Ausgabeoperation die Elemente im Array angeben. Die eine Indexvariable wird für das Lesen, die andere für das Schreiben verwendet. Nach jeder Ein- bzw. Ausgabe wird der Index um eins erhöht.

Die Umwandlung ist semantisch *nicht* korrekt. Das Ein-/Ausgabeband wird in der symbolischen Analyse mit Hilfe von endlichen Arrays dargestellt und eine obere Schranke muß für die Ein-/Ausgabe gefunden werden. Wird sie falsch angegeben, ist die Umwandlung fehlerhaft.

Beispiel. Das Beispiel in Abbildung 3.28 liest die Anzahl der Elemente, die geschrieben werden sollen. Danach werden alle Zahlen von 1 bis zur Anzahl der zuvor eingegeben Elemente ausgegeben. Für die Umwandlung der E/A-Operationen werden zwei Indizes *ip* und *op* benötigt. Beide Indizes werden zu Beginn des Programms auf eins gesetzt.

Da es im Programm nur einen Datentyp gibt, werden nur E/A-Arrays für den Typ [1..100] deklariert. Die erste Leseanweisung wird durch die Operation $a := si[ip]; ip := ip + 1$ ersetzt. Der Wert, der an der Stelle *ip* im Array *si* steht,

entspricht der ersten Eingabe. Der Zeiger ip wird nach der Leseoperation um eins erhöht. In der For-Schleife wird die Write-Anweisung durch die Zuweisungen $so[op] := i; op := op + 1$ substituiert.

Das Programm ist nur bis 1000-Ausgaben definiert. Jede weitere Ausgabe überschreitet den Definitionsbereich von op und so.

```
decl a:[1..100];

read a;
for i:=1 to a
    write i
```

wird zu

```
decl a:[1..100];
     ip:[1..1000];
     op:[1..1000];
     si:array[1..1000] of [1..100];
     so:array[1..1000] of [1..100];

ip:=1;
op:=1;
a:=si[ip];
ip:=ip+1;
for i:=1 to a
    so[op]:=i;
    op:=op+1
```

Abbildung 3.28: Beispiel mit Ein-/Ausgabe

Kapitel 4

KONKLUSION

Die symbolische Analyse ist ein mächtiges Werkzeug. Sie ist vielseitig anwendbar. Insbesondere im Bereich der symbolischen Verifikation von Echtzeitprogrammen ist sie ein wichtiges Hilfsmittel.

Eine Implementierung der Wert-konditionierenden Methode in Mathematica (vgl. [63]) hat vielversprechende Ergebnisse gebracht. Leider fehlen Analysen für Pointer, komplexe Datenstrukturen, Objekte und Anweisungen, wie GOTOs und Schleifenabbruchanweisungen. Diese Erweiterungen machen die Analysen noch aufwendiger und komplizierter.

Symbolische Evaluierung wird in den nächsten Jahren ein wichtiges Thema in der Software-Entwicklung sein: Der Computer kann selbst Programme analysieren. Diese Aufgabe war ursprünglich den Menschen vorbehalten. Jedoch kann durch eine geeignete Theorie und mit komplexen Algebrawerkzeugen der Prozeß des Analysierens automatisiert werden.

Literatur

[1] A. V. Aho, R. Sethi und J. D. Ullmann, *Compilerbau*, Addison-Wesley Verlag Deutschland, 1988.

[2] V. Ambriola, F. Giannoti, D. Pedreschi und F. Turini, *Symbolic semantics and program reduction*, IEEE Trans. Software Eng. **11** (1985), no. 9, 784–794.

[3] R. Arnold, F. Mueller, D. Whalley und M. Harmon, *Bounding worst-case instruction cache performance*, Proc. of the Fifteenth IEEE Real-Time Systems Symposium (1994), 172–181.

[4] J. Bergeretti und B. A. Carre, *Information flow and data-flow analysis of while-programs*, ACM Transactions on Programming Language **7** (1985), no. 1, 27–61.

[5] J. Blieberger, *Discrete loops and worst case performance*, Comput. Lang. **20** (1994), no. 3, 193–212.

[6] J. Blieberger und R. Lieger, *Worst-case space and time complexity of recursive procedures*, Real-Time Systems (1996), 115–144.

[7] W. Blume und R. Eigenmann, *Demand-driven, symbolic range propagation*, Proc. of the 8th Workshop on Lang. and Compilers for Parallel Computing (1995).

[8] F. Bourdoncle, *Interprocedural abstract interpretation of block structured languages with nested procedures aliasing and recursivity*, PLILP'90 **456 of LNCS** (1990), 307–323.

[9] ———, *Abstract debugging of higher-order imperative languages*, ACM-SIGPLAN-PLDI-6/93/Albuquerque (1993), 46–55.

[10] M. Brockhaus, T. Berger und A. Ertl, *Übersetzerbau*, Skriptum zur Vorlesung, Institut für Computersprachen, Abteilung für Übersetzerbau, Technische Universität Wien, 1994.

[11] R. Chapman, *Symbolic execution, proof, and timing analysis of spark ada*, Tech. Report DCSC/TR/93/16, University of York, Dependable Computing Systems Centre, Oct 1993.

[12] ———, *Static timing analysis and program proof*, Ph.D. thesis, Univ. of York, Computer Sience, 1995.

[13] T. E. Cheatham, H. Holloway und J. A. Townley, *Symbolic evaluation and the analysis of programs*, IEEE Trans. Software Eng. **5** (1979), no. 4, 402–317.

[14] A. Cimitile, A. D. Lucia und M. Munro, *Qualifying reusable functions using symbolic execution*, Proceedings of 2nd IEEE Working Conference on Reverse Engineering (Toronto, Canada), Soc. Press, pp. 178–187.

[15] A. Cimitile, A. D. Lucia und M. Munro, *A specification driven slicing process for identifying reuseable functions*, Tech. Report 3/95, Dep. of Computer Science, University of Durham, South Road, DH1 3LE Durham, UK, 1995.

[16] Clarke und A. Lori, *A method combining testing and verification*, IEEE Trans. Software Eng. **11** (1985), no. 12, 215–222.

[17] L. A. Clarke und D. J. Richardson, *Symbolic evaluation methods for program analysis*, Program Flow Analysis (Englewoord Cliffs, New Jersey) (S. S. Muchnick und N. D. Jones, eds.), Prentice Hall, Englewoord Cliffs, New Jersey, 1981, pp. 264–300.

[18] A. Coen-Porsini, F. DePaoli, C. Ghezzi und D. Mandrioli, *Software specialization via symbolic execution*, IEEE Trans. Software Eng. **17** (1991), no. 9, 884–899.

[19] P. Cousot und R. Cousot, *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximating of fixpoints*, Proc. of the 4th ACM Symp. on POPL (1977), 238–252.

[20] ———, *Abstract interpretation and application to logic programs*, The Journal of Logic Programming (1992), 103–171.

[21] T. Fahringer, *Toward Symbolic Performance Prediction of Parallel Programs*, IEEE Proc. of the 1996 International Parallel Processing Symposium (Honolulu, Hawaii), pp. 474–478.

[22] ———, *Symbolic Expression Evaluation to Support Parallelizing Compilers*, to appear in: IEEE Proc. of the 5th Euromicro Workshop on Parallel and Distributed Processing, London, UK.

[23] P. Flajolet, *Mathematical tools for automatic program analysis*, Tech. Report 603, Institute National de Recherche en informatique et en Automatique, 1987.

[24] P. Flajolet und B. Salvy, *Computer algebra libraries for combinatorial structures*, Tech. Report 2497, Institute National de Recherche en informatique et en Automatique, 1995.

[25] P. Flajolet, B. Salvy und P. Zimmermann, *Lambda-upsilon-omega the 1989 cookbook*, 1989.

[26] ———, *Automatic average-case analysis of algorithms*, Tech. Report 1233, Institute National de Recherche en informatique et en Automatique, 1990.

[27] M. P. Gerlek, E. Stoltz und M. Wolfe, *Beyond induction variables: Detecting and classifying sequences using a demand driven ssa form*, Transactions on Programming Languages (TOPLAS) **17** (1995), no. 1.

[28] M. J. C. Gordon, *The denotational description of programming languages*, Springer-Verlag New York, 1979.

[29] D. H. Greene und D. E. Knuth, *Mathematics for the analysis of algorithms*, Birkhäuser Boston, 1982.

[30] V. H. Haase, *Real-time behavior of programs*, IEEE Trans. on Softw. Eng. **7** (1981), no. 5, 494–501.

[31] M. R. Haghighat und C. D. Polychronopoulos, *Symbolic analysis for parallelizing compilers*, Tech. Report CSRD Report No. 1355, Center for Supercomputing Research and Development, Univ. of Illinois at Urbana Champaign, 415 CSRL - 1308 West Main Street, 1994.

[32] P. Havlak, *Interprocedural symbolic analysis*, Ph.D. thesis, RICE University, 1994.

[33] C. A. Healy, D. B. Whalley und M. G. Harmon, *Integrating the timing analysis of pipelining and instruction caching*, Proc. of the Sixteenth IEEE Real-Time Systems Symposium (1995), 288–297.

[34] J. E. Hopcroft und J. D. Ullman, *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie*, Addison-Wesley Verlag Deutschland, 1990.

[35] ISO/IEC 8652, *Ada reference manual*, 1995.

[36] C. Jordan, *Calculus of finite differences*, Chelsea, New York, 1965.

[37] B. W. Kernighan und D. M. Ritchie, *Programmieren in C*, Carl Hanser Verlag München Wien, 1983.

[38] J. King, *Symbolic execution and program testing*, Commun. ACM **19** (1976), no. 7, 385–394.

[39] L. Lamport, *Time, clocks and the ordering of events in a distributed system*, Commun. ACM **21** (1978), 558–565.

[40] K. G. Larsen, P. Petterson und W. Yi, *Compositional and symbolic model-checking of real-time systems*, Proc. of the Sixteenth IEEE Real-Time Systems Symposium (1995), 76–87.

[41] A. Leitsch, *Mathematische Logik und logikorientierten Programmiersprachen*, Skriptum zur Vorlesung, Institut für Computersprachen, Abteilung für Anwendung der formalen Logik, Technische Universität Wien, 1993.

[42] ———, *Algorithmen-, Rekursions- und Komplexitätstheorie*, Skriptum zur Vorlesung, Institut für Computersprachen, Abteilung für Anwendung der formalen Logik, Technische Universität Wien, 1994.

[43] A. Leitsch und G. Salzer, *Einführung in die Theorie der Informatik*, Skriptum zur Vorlesung, Institut für Computersprachen, Abteilung für Anwendung der formalen Logik, Technische Universität Wien, 1992.

[44] S. S. Lim, Y. H. Bae, G. T. Jang, B. D. Rhee, S. L. Min, C. Y. Park, H. Shin und C. S. Kim, *Worst case timing analysis of risc processors: R3000/r3010 case study*, Proc. of the Sixteenth IEEE Real-Time Systems Symposium (1995), 308–319.

[45] ———, *An accurate worst case timing analysis technique for risc processors*, Proc. of the Fifteenth IEEE Real-Time Systems Symposium (1994), 97–108.

[46] J. Liu und H. Lee, *Deterministic upperbounds of the worst-case execution times of cached programs*, Proc. of the Fifteenth IEEE Realt-Time Systems Symposium (1994), 182–191.

[47] S. S. Muchnick und N. D. Jones, *Abstract interpretation*, Program Flow Analysis (Englewoord Cliffs, New Jersey), Prentice Hall, Englewoord Cliffs, New Jersey, 1981, pp. 264–300.

[48] C. Park, *Predicting program execution times by analyzing static and dynamic program paths*, The Journal of Real-Time Systems **5** (1993), 31–62.

[49] W. Pugh, *Counting solutions to presburger formulas: How and why?*, Tech. Report UMIACS-TR-94-27, CS-TR-3234, Institute of Advanced Computer Studies, Dept. of Computer Sience, Univ. of Maryland, College Park, MD 20742, 1994.

[50] P. Puschner und C. Koza, *Calculating the maximum execution time of real-time programs*, The Journal of Real-Time Systems **1** (1989), 159–176.

[51] M. Rosendahl, *Automatic complexity analysis*, Functional Programming Languages and Computer Architectures, Conference Proceedings (1989), 144–156.

[52] A. Rosskopf, *Use of a static analysis tool for safety-critical ada applications*, Proc. of Ada-Europe'96 (Montreux), LNCS, pp. 183–197.

[53] S. Russel und P. Norvig, *Artificial intelligence - a modern approach*, Prentice Hall, Englewood Cliffs, New Jersey 07632, 1995.

[54] D. Sands, *A naive time analysis and its theory of cost equivalence*, J. Logic Computat. **5** (1995), no. 4, 495–541.

[55] A. V. Schedl, *Zeitanalyse von Echtzeitprogrammen mittels linearer Optimierung*, Master's thesis, Technische Universität Wien, 1993.

[56] A. C. Shaw, *Reasoning about time in higher-level language*, IEEE Trans. on Softw. Eng. **15** (1989), no. 7, 875–889.

[57] A. M. Stavely, *Verifying definite iteration over data structures*, IEEE Trans. on Softw. Eng. **21** (1995), no. 6, 507–514.

[58] Y. Steven und S. Malik, *Performance analysis of embedded software using implicit path enumeration*, ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (1995), 88–98.

[59] Y. Steven, S. Malik und A. Wolfe, *Efficient microachitecture modeling and path analysis for realtime-software*, Proc. of the Sixteenth IEEE Real-Time Systems Symposium (1995), 298–307.

[60] H. Stöcker, *Taschenbuch mathematischer Formeln und moderner Verfahren*, Verlag Harri Deutsch, 1992.

[61] P. Tu und D. Padua, *Gated ssa-based demand-driven symbolic analysis of parallelizing compilers*, ICS '95 Barcelona, Spain (1995).

[62] M. Wolfe, *Beyond induction variable*, ACM SIGPLAN'92 PLDI-6/92/CA (1992).

[63] S. Wolfram, *Mathematica — a system for doing mathematics by computer*, Addion Wesley Publishing Company, 1988.

[64] M. Young und R. Taylor, *Combining static concurrency analysis with symbolic execution*, IEEE Trans. Software Eng. **14** (1988), 1499–1511.

[65] H. Zima und B. Chapman, *Supercompilers for parallel and vector computers*, ACM Press, New York, 1991.