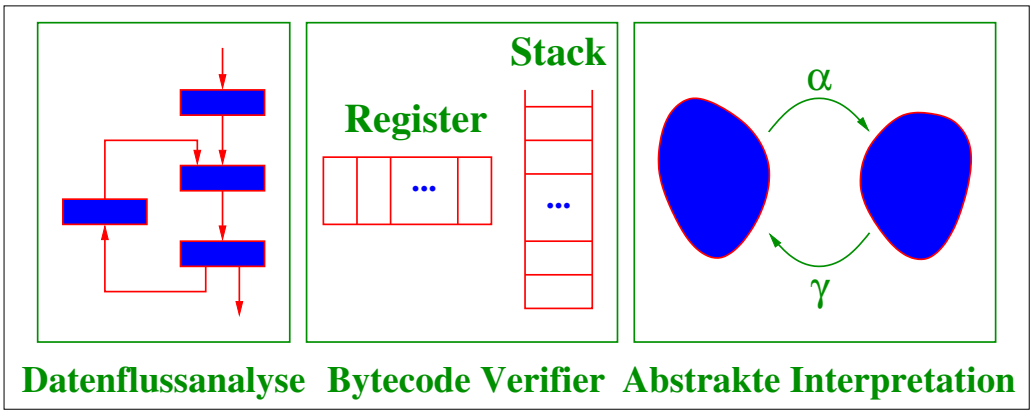


Modellierung, Analyse, Verifikation (Programmanalyse)

Skript zur Vorlesung im Wintersemester 2018/19 an der Universität Duisburg-Essen

Barbara König

30. Oktober 2018

Inhaltsverzeichnis

Kapitel 1

Einleitung

Zur automatischen Verifikation und Validierung von Programmen und Systemen benötigt man Verfahren, die bei Eingabe eines Programms und einer zugehörigen Spezifikation entscheiden, ob das Programm diese Spezifikation erfüllt. Dazu sagt allerdings der erste Satz von Rice, dass es unentscheidbar ist, ob die Funktion, die von einer Turingmaschine M berechnet wird, eine nicht-triviale Eigenschaft P hat. Nicht-trivial bedeutet dabei, dass P weder die leere Menge noch die Menge aller Funktionen ist. Ein Spezialfall dieses Satzes ist die Unentscheidbarkeit des Halteproblems. Und daraus folgt auch, dass das Verifikationsproblem für beliebige Programme unentscheidbar ist.

Es gibt jedoch effiziente Verifikations-Verfahren für endliche oder eingeschränkte Systeme und Programmklassen. [CGP00, Esp97]. Allerdings gibt es viele sicherheitskritische Programme, die diese Einschränkungen nicht erfüllen. Auch sie können analysiert werden, wenn man nicht-vollständige Verfahren zulässt. Man verlangt, dass diese Analyseverfahren niemals ein fehlerhaftes Programm als korrekt ansehen, es ist aber zulässig, korrekte Programme abzulehnen (einseitiger Irrtum). Auf diese Weise kann immer noch eine große Menge von Programmen analysisert und ihre Korrektheit verifiziert werden.

Graphisch ist diese Art des einseitigen Irrtums, auch Überapproximation genannt, in Abbildung 1.1 dargestellt.

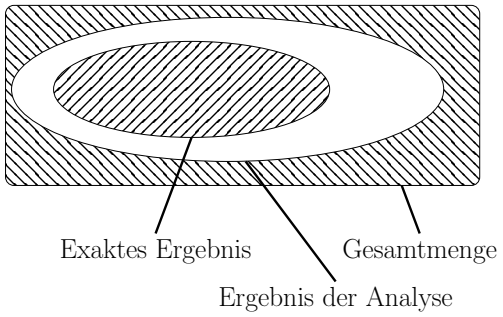


Abbildung 1.1: Einseitiger Irrtum

In der Vorlesung werden grundlegende Verfahren zur Programmanalyse, wie Datenflussanalyse, abstrakte Interpretation und (falls die Zeit noch reicht) Typsysteme vorgestellt und ihre Anwendung verdeutlicht. Neben der Anwendung solcher Verfahren zur Verifikation und Validierung von Programmen, ist ein wichtiges (und sogar das ursprüngliche) Einsatzgebiet die automatische Optimierung von Programmen durch Compiler.

In den letzten Jahren gab es auf dem Gebiet der Programmanalyse eine rege Forschungstätigkeit. Daher wird sich ein Teil dieses Skripts mit neueren Ergebnissen, auch aus dem Bereich der Analyse nebenläufiger Systeme beschäftigen.

Ziel aller vorgestellter Methoden ist dabei die automatische Analyse von Programmen und Systemen, die allein durch die Analyse des Programmtextes und nicht durch Ausführung des Programms erfolgt. Daher nennt man diese Verfahren oft auch *statische Analyse*, im Gegensatz zur *Laufzeit-Analyse* von Programmen. Es geht uns allein um die semantische Analyse von Programmen, v.a. in Hinblick auf deren Korrektheit, die Laufzeit-Komplexität bzw. Effizienz eines Algorithmus wird im weiteren nicht betrachtet. Ebensowenig beschäftigt sich dieses Skript mit der Laufzeit-Analyse, z.B. dem Testen oder dem Erstellen von Benchmarks.

Anwendungsmöglichkeiten der vorgestellten Techniken sind die automatische Optimierung bei der Programmübersetzung (Compilerbau), indem beispielsweise die wiederholte Berechnung von Ausdrücken vermieden wird oder toter Code entfernt wird. Des weiteren finden diese Techniken bei der Programmverifikation Anwendung, ein klassisches Beispiel ist der Java Bytecode Verifier. Vielversprechend ist auch der Einsatz für die Analyse reaktiver und nebenläufiger System, wie beispielsweise Netzwerk-Protokolle, bei denen der wechselseitige Ausschluss oder die korrekte (und geheime) Übertragung von Daten gewährleistet werden soll.

Dieses Skript basiert in Teilen auf [NNH99]. An Mathematik benötigt man vor allem Grundlagenwissen über partielle Ordnungen und Fixpunkttheorie (siehe Anhang B). Des weiteren werden wir im allgemeinen die Semantik [Win93] der verwendeten Programmiersprachen definieren, um Beweise über die Korrektheit der verwendeten Verfahren führen zu können.

Eine Analysetechnik, auf die im Rahmen dieser Vorlesung nicht eingegangen wird, ist die sogenannte Constraint-basierte Analyse. Informationen darüber finden sich in [NNH99].

Bitte schicken Sie Kommentare und Berichte über entdeckte Fehler und Ungenauigkeiten an Barbara König (`barbara_koenig@uni-due.de`).

Kapitel 2

Notation

Mengen: Mit $\{a \mid P(a)\}$ bezeichnen wir, wie üblich, die Menge aller Elemente a, die die Bedingung P erfüllen.

Mit $\mathbb{N}_0 = \{0, 1, 2, \ldots\}$ bezeichnen wir die natürlichen Zahlen mit Null, mit $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ die ganzen Zahlen und mit $\mathbb{R}$ die Menge aller reellen Zahlen.

$\mathcal{P}(A)$ bezeichnet die Potenzmenge, d.h. die Menge aller Teilmengen, einer Menge A. Beispielsweise gilt

$$\mathcal{P}(\{a, b, c\}) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}.$$

Funktionen: Sei $f\colon A \to B$ eine Funktion. Eine solche Funktion kann auch als Menge von Paaren $\{(a, f(a)) \mid a \in A\} \subseteq A \times B$ aufgefasst werden. Vor allem wenn A endlich ist, kann man die Funktion f so explizit angeben. Um zu zeigen, dass es sich dabei um eine Funktion handelt, schreiben wir manchmal auch $[a_1 \mapsto f(a_1), \ldots, a_n \mapsto f(a_n)]$, falls $A = \{a_1, \ldots, a_n\}$.

Mit $A \to B$ wird gelegentlich die Menge aller Funktionen, die von A nach B abbilden, bezeichnet.

Falls $f\colon A \to B$ eine Funktion ist und $a \in A$, $b \in B$ gilt, so bezeichnen wir mit $f[a \mapsto b]$ eine Funktion, die auf allen Werten aus A—abgesehen von a—mit f identisch ist. Außerdem wird a auf b abgebildet. D.h.

$$(f[a \mapsto b])(a') = \begin{cases} f(a') & \text{falls } a' \neq a \\ b & \text{sonst} \end{cases}$$

Mit $f\colon A \dashrightarrow B$ bezeichnen wir eine partielle Funktion, die nicht notwendigerweise auf jedem Element von A definiert ist.

Listen: Sei A eine Menge, dann bezeichnen wir mit A^* die Menge aller Listen (oder auch Sequenzen über A). Eine Liste $l \in A^*$ schreiben wir $a_1 \ldots a_n$ oder auch $[a_1, \ldots, a_n]$, für den Fall, dass die Elemente klar voneinander abgetrennt werden sollen. Mit $\circ$ bezeichnen die Konkatenation von Listen. Des weiteren benötigen wir folgende partielle Funktionen auf Listen:

$$\begin{aligned}
\mathit{first}\colon A^* &\dashrightarrow A & \mathit{first}(a_1 \ldots a_n) &= a_1 \\
\mathit{rest}\colon A^* &\dashrightarrow A^* & \mathit{rest}(a_1 a_2 \ldots a_n) &= a_2 \ldots a_n \\
_!_\colon A^* \times \mathbb{N}\backslash\{0\} &\dashrightarrow A & (a_1 \ldots a_{i-1} a_i\, a_{i+1} \ldots a_n)!i &= a_i \\
_[_ \mapsto _]\colon A^* \times \mathbb{N}\backslash\{0\} \times A &\to A^* & (a_1 \ldots a_{i-1} a_i\, a_{i+1} \ldots a_n)[i \mapsto b] &= a_1 \ldots a_{i-1} b\, a_{i+1} \ldots a_n
\end{aligned}$$

Operatoren: Sei $\otimes\colon A \times A \to A$ ein assoziativer und kommutativer Operator. Die "große" Variante $\bigotimes$ dieses Operators wird auf verschiedene Arten und Weisen gebraucht:

$$\begin{aligned}
\bigotimes_{i=1}^{n} a_i &= a_1 \otimes \ldots \otimes a_n \\
\bigotimes_{i \in I} a_i &= a_{i_1} \otimes \ldots \otimes a_{i_n} \qquad \text{falls } I = \{i_1, \ldots, i_n\} \\
\bigotimes \{a_1, \ldots, a_n\} &= a_1 \otimes \ldots \otimes a_n
\end{aligned}$$

Zum Beispiel:

$$\bigcup \{A_i \mid i \in \{1, \ldots, n\}\} = \bigcup_{i \in \{1,\ldots,n\}} A_i = \bigcup_{i=1}^{n} A_i = A_1 \cup \ldots \cup A_n.$$

Im Laufe des Skript werden häufig die Operatoren $\sqcup$ (Supremum) und $\sqcap$ (Infimum) benutzt (siehe Anhang B).

Programme und Variablenbelegungen: Ein Tupel der Form $\langle S, \sigma \rangle$ steht im folgenden für ein Programmstück S mit Variablenbelegung σ (siehe Anhang A). Programme sind zumeist in Schreibmaschinenschrift gesetzt.

Schlussregeln: Typregeln oder andere Schlussregel werden zumeist in der Form

$$\frac{P}{Q}$$

angegeben. Dies bedeutet, wenn die Vorbedingung (oder Prämisse) P erfüllt ist, dann kann man daraus die Folgerung Q ziehen. Beispielsweise könnte man schreiben:

$$\frac{x\colon \mathit{int} \quad y\colon \mathit{int}}{x + y\colon \mathit{int}}$$

Das heißt, falls beide Ausdrücke x und y den Typ *int* (Integer) haben, so hat auch $x + y$ den Typ *int*.

Kapitel 3

Datenflussanalyse

Datenflussanalyse beschäftigt sich damit, den "Fluss" von Daten durch ein Programm zu verfolgen. Die daraus gewonnenen Informationen können beispielsweise von einem Compiler zur Programmoptimierung eingesetzt werden, etwa um unnütze Befehle zu entfernen oder die wiederholte Berechnung komplexer Ausdrücke zu vermeiden. Die Ergebnisse der Datenflussanalyse müssen dabei nicht immer ganz exakt sein, das Verfahren darf sich aber immer nur in eine Richtung irren. D.h., es ist erlaubt, Daten zu liefern, die dazu führen, dass ein überflüssiger Befehl *nicht* gestrichen wird, es darf jedoch auf keinen Fall ein Befehl, der Einfluss auf den weiteren Programmablauf hat, gestrichen werden.

Es gibt sehr viele verschiedene Arten von Datenflussanalyse, einige davon werden wir im folgenden kennenlernen. Diese Analysen haben jedoch gemeinsame Eigenschaften, so dass man sie unter dem Oberbegriff der monotonen Frameworks zusammenfassen kann. Im Rahmen dieser monotonen Frameworks kann man auch ein allgemeines Verfahren angeben, das das Ergebnis einer Datenflussanalyse bestimmt (Worklist-Algorithmus).

Das folgende Kapitel basiert im wesentlichen auf [NNH99].

3.1 Beispiel: Analyse verfügbarer Ausdrücke

Wir beginnen mit einem kleinen Beispiel und analysieren folgendes unten angegebenes Programmstück S. Die eckigen Klammern mit hochgestellten Zahlen bezeichnen dabei die einzelnen Programmblöcke, die wir durchnumerieren, um besser auf sie verweisen zu können.

$$[\texttt{x:=a+b}]^1\texttt{;}[\texttt{y:=a*b}]^2\texttt{;while }[\texttt{y>a+b}]^3\texttt{ do }[\texttt{a:=a+1}]^4\texttt{; }[\texttt{x:=a+b}]^5\texttt{ od;}[\texttt{z:=x}]^6$$

Dieses Programm kann auch graphisch als Datenflussdiagramm dargestellt werden, das beschreibt, in welcher Reihenfolge die einzelnen Blöcke durchlaufen werden können. In diesem Fall sieht das aus wie in Abbildung 3.1. Alternativ zur graphischen Notation kann man dieses Diagramm auch durch eine sogenannte Flussrelation $\mathit{flow}(S) = \{(1,2),(2,3),(3,4),(4,5),(5,3),(3,6)\}$ beschreiben, die der Kantenmenge des abgebildeten Graphen entspricht. Die Tatsache, dass der Übergang vom Block 3 aus von der Belegung von `a`, `b` und `y` abhängt, wird hier einfach vernachlässigt. (Siehe auch die obigen Aussagen über einseitigen Irrtum.) Außerdem bezeichnet $\mathit{init}(S) = 1$ den initialen Block und $\mathit{final}(S) = \{6\}$ die Menge der finalen Blöcke (hier gibt es nur einen finalen Block).

Wir wollen nun für den Eingang und Ausgang jedes Blocks folgende Menge bestimmen:

> Die Menge aller arithmetischen Ausdrücke, die auf auf allen Pfaden bis zu diesem Programmpunkt bereits berechnet wurden und nicht zwischendurch modifiziert wurden.

In unserem Beispiel steht der Ausdruck `a+b` am Eingang zum Block 1 nicht zur Verfügung, da er noch nicht berechnet wurde, wohl aber am Eingang zu Block 2. Am Ausgang zu Block 4 und Eingang zu Block 5 steht er allerdings auch nicht zur Verfügung, da in Block 4 die Variable a

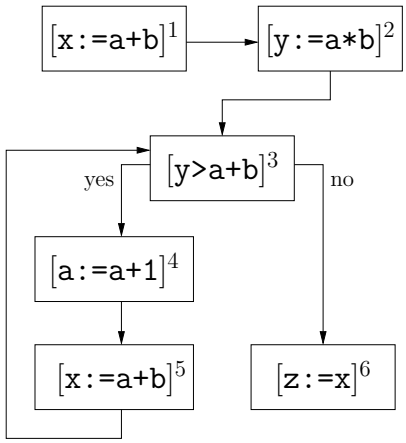


Abbildung 3.1: Ein Datenflussdiagramm

verändert wurde. Insgesamt erwarten wir das Ergebnis in Tabelle 3.1, kennen aber noch kein automatisches Verfahren zu seiner Berechnung. Dabei bezeichnen $\mathsf{A}_\circ(\ell)$ und $\mathsf{A}_\bullet(\ell)$ die Analysewerte, d.h. die Menge der verfügbaren Ausdrücke (available expressions) am Eingang bzw. Ausgang des Blocks ℓ.

ℓ	$\mathsf{A}_\circ(\ell)$	$\mathsf{A}_\bullet(\ell)$
1	$\emptyset$	{a+b}
2	{a+b}	{a+b, a*b}
3	{a+b}	{a+b}
4	{a+b}	$\emptyset$
5	$\emptyset$	{a+b}
6	{a+b}	{a+b}

Tabelle 3.1: Gewünschtes Ergebnis der Analyse verfügbarer Ausdrücke

Da also der Ausdruck `a+b` am Eingang zu Block 3 auf jeden Fall verfügbar ist, könnte ein Compiler den Maschinencode, in den das Programm übersetzt wird, dadurch optimieren, dass er die Ausdruck `a+b` bei seiner Berechnung in Block 1 bzw. 6 in einem eigenen Register zwischenspeichert, so dass der Ausdruck in Block 3 nicht neu berechnet werden muss. In Block 5 darf jedoch aufgrund der vorher stattfindenden Zuweisung an die Variable `a` *nicht* der zwischengespeicherte Wert verwendet werden.

Um Tabelle 3.1 aus dem Programm S herausrechnen zu können, definieren wir zunächst, welche Ausdrücke am Ende eines Blocks ℓ entfernt ($\mathit{kill}(\ell)$) und welche hinzugefügt ($\mathit{gen}(\ell)$) werden müssen. Dabei gehen wir von der Syntax und Semantik von WHILE-Programmen aus, wie sie in Anhang A definiert wird. Insbesondere gibt es dabei drei Typen von Blöcken: Zuweisungsblöcke der Form $[\texttt{x:=}a]^\ell$, skip-Anweisungen der Form $[\texttt{skip}]^\ell$ und Boole'sche Ausdrücke $[b]^\ell$. Des weiteren bezeichnen $\mathbf{AExp}_*$, die Menge aller nicht-trivialen arithmetischen Ausdrücke, die in dem untersuchten Programmstück S vorkommen, $\mathbf{AExp}(a)$ die arithmetischen Ausdrücke und $\mathit{Var}(a)$ die Variablen die in einem Ausdruck $a \in \mathbf{AExp}_*$ vorkommen. Mit $\mathit{blocks}(S)$ bezeichnet man die Menge der in S vorkommenden Blöcke. Außerdem bezeichnen wir mit $\mathit{block}_S(\ell)$ bzw. $\mathit{block}(\ell)$ die Anweisung innerhalb von Block ℓ (`x:=`a, `skip` oder b).

$$
\begin{array}{rcl}
\mathit{kill}(\ell) & = & \begin{cases} \{a' \in \mathbf{AExp}_* \mid \mathtt{x} \in \mathit{Var}(a')\} & \text{falls } \mathit{block}(\ell) = (\mathtt{x:=}a) \\ \emptyset & \text{sonst} \end{cases} \\
\mathit{gen}(\ell) & = & \begin{cases} \{a' \in \mathbf{AExp}(a) \mid \mathtt{x} \notin \mathit{Var}(a')\} & \text{falls } \mathit{block}(\ell) = (\mathtt{x:=}a) \\ \emptyset & \text{falls } \mathit{block}(\ell) = \mathtt{skip} \\ \mathbf{AExp}(b) & \text{falls } \mathit{block}(\ell) = b \end{cases}
\end{array}
$$

In unserem Fall sehen die Werte von $\mathit{kill}(\ell)$ und $\mathit{gen}(\ell)$ aus wie in Tabelle 3.2.

ℓ	$\mathit{kill}(\ell)$	$\mathit{gen}(\ell)$
1	$\emptyset$	{a+b}
2	$\emptyset$	{a*b}
3	$\emptyset$	{a+b}
4	{a+b,a*b,a+1}	$\emptyset$
5	$\emptyset$	{a+b}
6	$\emptyset$	$\emptyset$

Tabelle 3.2: $\mathit{kill}(\ell)$ und $\mathit{gen}(\ell)$

Allgemein kann man die Mengen $\mathsf{A}_\circ(\ell)$ und $\mathsf{A}_\bullet(\ell)$ durch ein Gleichungssystem beschreiben. Dabei werden für $\mathsf{A}_\circ(\ell)$ die (Ausgangs-)Analyseergebnisse aller Vorgängerblöcke mit Hilfe eines Schnitts zusammengefasst. Der Schnitt muss verwendet werden, da wir ja nach den Ausdrücken suchen, die auf allen Pfaden bereits berechnet wurden. Die Menge $\mathsf{A}_\bullet(\ell)$ bestimmt man, indem man aus $\mathsf{A}_\circ(\ell)$ die Ausdrücke der Menge $\mathit{kill}(\ell)$ entfernt und die Ausdrücke der Menge $\mathit{gen}(\ell)$ hinzufügt.

$$
\begin{array}{rcl}
\mathsf{A}_\circ(\ell) & = & \begin{cases} \emptyset & \text{falls } \ell = \mathit{init}(S) \\ \bigcap\{\mathsf{A}_\bullet(\ell') \mid (\ell', \ell) \in \mathit{flow}(S)\} & \text{sonst} \end{cases} \\
\mathsf{A}_\bullet(\ell) & = & (\mathsf{A}_\circ(\ell) \backslash \mathit{kill}(\ell)) \cup \mathit{gen}(\ell).
\end{array}
$$

Insgesamt erhält man folgende Gleichungen:

$$
\begin{array}{rcl}
\mathsf{A}_\circ(1) & = & \emptyset \\
\mathsf{A}_\circ(2) & = & \mathsf{A}_\bullet(1) \\
\mathsf{A}_\circ(3) & = & \mathsf{A}_\bullet(2) \cap \mathsf{A}_\bullet(5) \\
\mathsf{A}_\circ(4) & = & \mathsf{A}_\bullet(3) \\
\mathsf{A}_\circ(5) & = & \mathsf{A}_\bullet(4) \\
\mathsf{A}_\circ(6) & = & \mathsf{A}_\bullet(3)
\end{array}
\qquad
\begin{array}{rcl}
\mathsf{A}_\bullet(1) & = & \mathsf{A}_\circ(1) \cup \{\mathtt{a}+\mathtt{b}\} \\
\mathsf{A}_\bullet(2) & = & \mathsf{A}_\circ(2) \cup \{\mathtt{a}*\mathtt{b}\} \\
\mathsf{A}_\bullet(3) & = & \mathsf{A}_\circ(3) \cup \{\mathtt{a}+\mathtt{b}\} \\
\mathsf{A}_\bullet(4) & = & \mathsf{A}_\circ(4) \backslash \{\mathtt{a}+\mathtt{b}, \mathtt{a}*\mathtt{b}, \mathtt{a}+1\} \\
\mathsf{A}_\bullet(5) & = & \mathsf{A}_\circ(5) \cup \{\mathtt{a}+\mathtt{b}\} \\
\mathsf{A}_\bullet(6) & = & \mathsf{A}_\circ(6)
\end{array}
$$

Da wir von möglichst vielen Ausdrücken wissen wollen, dass sie verfügbar sind, suchen wir nach der größten Lösung dieses Gleichungssystems in Bezug auf Inklusion. Allerdings ist dieses Gleichungssystem in gewissem Sinne "rekursiv" und man kann es nicht direkt durch Einsetzen lösen.

Aufgabe 3.1.1 Betrachten Sie folgendes Programmstück:

$$[\mathtt{x:=a+b}]^1\mathtt{;while}\ [\mathtt{true}]^2\ \mathtt{do}\ [\mathtt{skip}]^3\ \mathtt{od}$$

Das zugehörige Gleichungssystem zur Analyse verfügbarer Ausdrücke besitzt mehrere Lösungen. Bestimmen Sie die kleinste und die größte Lösung und untersuchen Sie, welche der beiden Lösungen mehr Information über das Programm liefert.

Man kann obiges Gleichungssystem auch als Funktion $F\colon \mathcal{P}(\mathbf{AExp}_*)^{12} \to \mathcal{P}(\mathbf{AExp}_*)^{12}$ auffassen, wobei

$$\begin{aligned} F(N_1,\ldots,N_6,X_1,\ldots,X_6) \;=\; & (\emptyset, X_1, X_2 \cap X_5, X_3, X_4, X_3,\\ & N_1 \cup \{\mathtt{a}+\mathtt{b}\}, N_2 \cup \{\mathtt{a}*\mathtt{b}\}, N_3 \cup \{\mathtt{a}+\mathtt{b}\},\\ & N_4 \backslash \{\mathtt{a}+\mathtt{b}, \mathtt{a}*\mathtt{b}, \mathtt{a}+1\}, N_5 \cup \{\mathtt{a}+\mathtt{b}\}, N_6) \end{aligned}$$

analog zu den oben angegebenen Gleichungen. Wir suchen nun den größten Fixpunkt von F. Wir wissen, dass F monoton ist, d.h., aus[1] $(X_1,\ldots,X_6,N_1,\ldots,N_6) \sqsubseteq (X'_1,\ldots,X'_6,N'_1,\ldots,N'_6)$ folgt $F(N_1,\ldots,N_6,X_1,\ldots,X_6) \sqsubseteq F(N'_1,\ldots,N'_6,X'_1,\ldots,X'_6)$. Außerdem ist bekannt dass $\mathcal{P}(\mathbf{AExp})^{12}$ ein vollständiger Verband ist und die Ordnung $\sqsubseteq$ die Descending Chain Condition erfüllt, denn $\mathcal{P}(\mathbf{AExp}_*)^{12}$ ist eine endliche Menge. Daraus folgt mit Hilfe der Fixpunktsätze (siehe Anhang B), dass der größte Fixpunkt durch iterierte Anwendung von F auf $(\mathbf{AExp}_*,\ldots,\mathbf{AExp}_*)$ erreicht werden kann. Es ist zu beachten, dass in unserem Beispiel $\mathbf{AExp}_* = \{\mathtt{a}+\mathtt{b}, \mathtt{a}*\mathtt{b}, \mathtt{a}+1\}$ gilt. Man erhält folgende Sequenz von Tupeln:

$$\begin{aligned}
F^0(\mathbf{AExp}_*,\ldots) &= (\mathbf{AExp}_*,\ldots,\mathbf{AExp}_*)\\
F^1(\mathbf{AExp}_*,\ldots) &= (\emptyset,\mathbf{AExp}_*,\mathbf{AExp}_*,\mathbf{AExp}_*,\mathbf{AExp}_*,\mathbf{AExp}_*,\mathbf{AExp}_*,\mathbf{AExp}_*,\mathbf{AExp}_*,\emptyset,\mathbf{AExp}_*,\mathbf{AExp}_*)\\
F^2(\mathbf{AExp}_*,\ldots) &= (\emptyset,\mathbf{AExp}_*,\mathbf{AExp}_*,\mathbf{AExp}_*,\emptyset,\mathbf{AExp}_*,\{\mathtt{a}+\mathtt{b}\},\mathbf{AExp}_*,\mathbf{AExp}_*,\emptyset,\mathbf{AExp}_*,\mathbf{AExp}_*)\\
F^3(\mathbf{AExp}_*,\ldots) &= (\emptyset,\{\mathtt{a}+\mathtt{b}\},\mathbf{AExp}_*,\mathbf{AExp}_*,\emptyset,\mathbf{AExp}_*,\{\mathtt{a}+\mathtt{b}\},\mathbf{AExp}_*,\mathbf{AExp}_*,\emptyset,\{\mathtt{a}+\mathtt{b}\},\mathbf{AExp}_*)\\
F^4(\mathbf{AExp}_*,\ldots) &= (\emptyset,\{\mathtt{a}+\mathtt{b}\},\{\mathtt{a}+\mathtt{b}\},\mathbf{AExp}_*,\emptyset,\mathbf{AExp}_*,\{\mathtt{a}+\mathtt{b}\},\{\mathtt{a}+\mathtt{b},\mathtt{a}*\mathtt{b}\},\mathbf{AExp}_*,\emptyset,\{\mathtt{a}+\mathtt{b}\},\mathbf{AExp}_*)\\
F^5(\mathbf{AExp}_*,\ldots) &= (\emptyset,\{\mathtt{a}+\mathtt{b}\},\{\mathtt{a}+\mathtt{b}\},\mathbf{AExp}_*,\emptyset,\mathbf{AExp}_*,\{\mathtt{a}+\mathtt{b}\},\{\mathtt{a}+\mathtt{b},\mathtt{a}*\mathtt{b}\},\{\mathtt{a}+\mathtt{b}\},\emptyset,\{\mathtt{a}+\mathtt{b}\},\mathbf{AExp}_*)\\
F^6(\mathbf{AExp}_*,\ldots) &= (\emptyset,\{\mathtt{a}+\mathtt{b}\},\{\mathtt{a}+\mathtt{b}\},\{\mathtt{a}+\mathtt{b}\},\emptyset,\{\mathtt{a}+\mathtt{b}\},\{\mathtt{a}+\mathtt{b}\},\{\mathtt{a}+\mathtt{b},\mathtt{a}*\mathtt{b}\},\{\mathtt{a}+\mathtt{b}\},\emptyset,\{\mathtt{a}+\mathtt{b}\},\mathbf{AExp}_*)\\
F^7(\mathbf{AExp}_*,\ldots) &= (\emptyset,\{\mathtt{a}+\mathtt{b}\},\{\mathtt{a}+\mathtt{b}\},\{\mathtt{a}+\mathtt{b}\},\emptyset,\{\mathtt{a}+\mathtt{b}\},\{\mathtt{a}+\mathtt{b}\},\{\mathtt{a}+\mathtt{b},\mathtt{a}*\mathtt{b}\},\{\mathtt{a}+\mathtt{b}\},\emptyset,\{\mathtt{a}+\mathtt{b}\},\{\mathtt{a}+\mathtt{b}\})\\
F^8(\mathbf{AExp}_*,\ldots) &= (\emptyset,\{\mathtt{a}+\mathtt{b}\},\{\mathtt{a}+\mathtt{b}\},\{\mathtt{a}+\mathtt{b}\},\emptyset,\{\mathtt{a}+\mathtt{b}\},\{\mathtt{a}+\mathtt{b}\},\{\mathtt{a}+\mathtt{b},\mathtt{a}*\mathtt{b}\},\{\mathtt{a}+\mathtt{b}\},\emptyset,\{\mathtt{a}+\mathtt{b}\},\{\mathtt{a}+\mathtt{b}\})
\end{aligned}$$

Wir haben damit den Fixpunkt erreicht, der genau den Werten in Tabelle 3.1 entspricht. Die obige Art der Fixpunktberechnung ist allerdings relativ ineffizient, wir werden später ein schnelleres Verfahren kennenlernen.

Aufgabe 3.1.2 Führen Sie eine Analyse der verfügbaren Ausdrücke für folgendes Programm durch.

```
[x:=(a-b)*a]^1;
[x:=x+1]^2;
if [x>0]^3
   then
      while [(a-b)>0]^4 do
         [b:=b+1]^5;
         [x:=a-b]^6
      od
   else [skip]^7
fi
```

Aufgabe 3.1.3 Bestimmen Sie ein WHILE-Programm mit einem Block $[\mathtt{x:=}a]^\ell$, so dass der arithmetische Ausdruck vor jedem Eintritt in diesen Block bereits berechnet wurde und sich seitdem nicht verändert hat, dies von der Analyse aber nicht erkannt wird.

[1] Dabei ist die Ordnung auf $\mathcal{P}(\mathbf{AExp})^{12}$ komponentenweise definiert, d.h., $(X_1,\ldots,X_6,N_1,\ldots,N_6) \sqsubseteq (X'_1,\ldots,X'_6,N'_1,\ldots,N'_6)$ steht hier für $X_1 \subseteq X'_1,\ldots,X_6 \subseteq X'_6, N_1 \subseteq N'_1,\ldots,N_6 \subseteq N'_6$.

3.2 Monotone Frameworks zur Datenflussanalyse

3.2.1 Grundlegende Definitionen

Im vorherigen Abschnitt haben wir eine *Vorwärts-Analyse* kennengelernt, bei der die einem Block zugeordnete Menge von Ausdrücken aus den Mengen der Vorgänger-Blocks berechnet wurde. Dies lag daran, dass wir an der Vergangenheit eines Blocks interessiert waren. Ist man jedoch an der Zukunft eines Blocks interessiert, d.h. an allen Pfaden, die von diesem Block aus möglich sind, so muss man eine sogenannte *Rückwärts-Analyse* durchführen. Des weiteren haben wir im vorherigen Beispiel den *Schnitt* verwendet, weil wir nur an solchen Informationen interessiert waren, die auf jeden Fall für alle Pfade gelten. Weil das Analyseergebnis umso genauer wurde, je größer die Menge war, haben wir nach dem *größten Fixpunkt* gesucht. Das ist durch die Verwendung des Schnitts bedingt. Manchmal ist man allerdings auch daran interessiert, ob es möglicherweise einen Pfad gibt, der eine bestimmte Eigenschaft hat. In diesem Fall verwendet man die *Vereinigung* und ist am *kleinsten Fixpunkt* interessiert.

Im folgenden werden wir Beispiele für solche Analysen kennenlernen, zunächst sind wir jedoch an einem allgemeinen Framework interessiert, in dem man solche Analysen zusammenfassen kann.

Ein Bestandteile der vorhergehenden Analyse waren die Teilmengen von $\mathbf{AExp}_*$, d.h., die Mengen verfügbarer Ausdrücke, zusammen mit Vereinigung und Schnitt. Dieses System von Mengen nennt man auch den *Property Space*. Auch wenn Mengen als Datentyp dafür recht geeignet erscheinen, wollen wir später den Blöcken eines Programms auch andere Elemente zuordnen, daher werden im folgenden nicht mehr über Systeme von Mengen, sondern, allgemeiner, über vollständige Verbände reden. Zudem fordern wir, dass die dem Verband zugrundeliegende Ordnung die Ascending bzw. die Descending Chain Condition erfüllt.

Ein zweiter Bestandteil waren Transferfunktionen, die beschrieben, wie das Analyseergebnis durch die Passage durch einen Block verändert wird. Bisher hatten diese Funktionen die Form $f_\ell(\mathsf{A}) = \mathsf{A} \backslash \mathit{kill}(\ell) \cup \mathit{gen}(\ell)$. Des weiteren sind diese Funktionen natürlicherweise monoton, d.h., aus $\mathsf{A} \subseteq \mathsf{A}'$ folgt $f_\ell(\mathsf{A}) \subseteq f_\ell(\mathsf{A}')$.

Definition 3.2.1 (Monotoner Framework) *Ein* monotoner Framework $(L, \mathcal{F})$ *besteht aus einem vollständigen Verband L (auch* Property Space *genannt), dessen Ordnung mit $\sqsubseteq$ und dessen Supremumsoperator (kleinste obere Schranke) mit $\sqcup$ bezeichnet wird. Des weiteren ist $\mathcal{F}$ eine Menge von monotonen Funktionen von L nach L, die die Identität enthält und die unter Funktionskomposition abgeschlossen ist.*

Die partielle Ordnung des Verbandes hat in diesem Zusammenhang folgende Bedeutung: falls a und b zwei Analyseergebnisse sind und es gilt $a \sqsubseteq b$, dann ist a genauer (oder auch besser) als b.

Für die konkrete Analyse eines Programms, benötigt man noch zusätzliche Komponenten, insbesondere den Flussgraph und die Transferfunktionen für jeden Block.

Definition 3.2.2 (Instanz eines monotonen Frameworks) *Die* Instanz eines monotonen Frameworks $(L, \mathcal{F})$ *ist ein Tupel $(L, \mathcal{F}, \mathbf{Lab}, F, E, \iota, f.)$, dessen Komponenten $\mathbf{Lab}, F, E, \iota, f.$ folgende Bedeutung haben:*

- $\mathbf{Lab}$ *ist eine endliche Menge von* Labels
- $F \subseteq \mathbf{Lab} \times \mathbf{Lab}$ *ist die* Flussrelation
- $E \subseteq \mathbf{Lab}$ *ist eine Menge von sogenannten* extremalen Labels
- $\iota \in L$ *ist der Wert, der jedem extremalen Label zugeordnet wird*
- $f.$ *bildet jedes Label $\ell \in \mathbf{Lab}$ auf eine Funktion $f_\ell \in \mathcal{F}$ ab.*

In dem Beispiel aus Abschnitt 3.1 sehen diese Komponenten folgendermaßen aus:

- Der vollständige Verband L ist $\mathcal{P}(\mathbf{AExp}_*)$, d.h., die Potenzmenge, der in dem Programm S vorkommenden arithmetischen Ausdrücke.

  Bei der Bestimmung der partiellen Ordnung müssen wir vorsichtig sein. In diesem Fall wird das Analyse-Ergebnis immer ungenauer, je kleiner die Menge wird. Daher verwenden wir $\supseteq$ als partielle Ordnung.

- Als Funktionsraum $\mathcal{F}$ könnte man z.B. die Menge aller monotonen Funktionen auf L verwenden.

- **Lab** steht in unserem Beispiel für die Menge aller im betrachteten Programm S vorkommenden Labels.

- Für die Flussrelation gilt $F = \mathit{flow}(S)$.

- Die extremalen Labels sind die Labels, bei denen die Analyse beginnt, in diesem Fall das initiale Label, es gilt also $E = \{\mathit{init}(S)\}$.

- In dieser Analyse wird dem initialen Label der Wert $\iota = \emptyset$ zugeordnet, da noch keine vorberechneten arithmetischen Ausdrücke zur Verfügung stehen.

- Die Funktion $f_\cdot$ ordnet jedem Label die (Transfer-)Funktion[2] $f_\ell \colon L \to L$ zu mit $f_\ell(l) = l \backslash \mathit{kill}(\ell) \cup \mathit{gen}(\ell)$.

Das eigentliche Ergebnis der Datenflussanalyse, $\mathsf{A}_\circ(\ell)$ und $\mathsf{A}_\bullet(\ell)$ für jeden Block ℓ wird dann folgendermaßen bestimmt: $\mathsf{A}_\circ, \mathsf{A}_\bullet \colon \mathbf{Lab}_* \to L$ sind die kleinste Lösung des folgenden Gleichungssystems:

$$\mathsf{A}_\circ(\ell) = \bigsqcup\{\mathsf{A}_\bullet(\ell') \mid (\ell', \ell) \in F\} \sqcup \iota^\ell_E \quad (3.1)$$

$$\text{wobei } \iota^\ell_E = \begin{cases} \iota & \text{falls } \ell \in E \\ \bot & \text{sonst} \end{cases}$$

$$\mathsf{A}_\bullet(\ell) = f_\ell(\mathsf{A}_\circ(\ell)) \quad (3.2)$$

Wir betrachten nun das Beispiel aus Abschnitt 3.1 und können zeigen, dass es eine Instanz eines monotonen Frameworks ist. Allerdings müssen wir an einer Stelle aufpassen: bei dem Beispiel haben wir die verschiedenen Datenflüsse mit Hilfe eines Schnitts zusammengefasst. In obiger Definition ist ein Supremum erforderlich. Dies ist jedoch kein Widerspruch, da der Schnitt das Supremum der Relation "$\supseteq$" ist, während die Vereinigung das Supremum von "$\subseteq$" ist. Wir müssen nur alle Operatoren sozusagen "umdrehen" und immer dort, wo $\sqcup$ steht, $\cap$ einsetzen. Damit ist die kleinste Lösung bezüglich $\supseteq$ die größte Lösung bezüglich $\subseteq$, und damit genau das, was wir benötigen.

Die obigen Gleichungen (3.1) und (3.2) werden formal als Fixpunktgleichungen aufgefasst. Wenn ein Programm m Labels enthält, dann kann man aufbauend auf den obigen Gleichungen eine Funktion $G \colon L^{2m} \to L^{2m}$ bestimmen, wie es in Abschnitt 3.1 beschrieben ist. Als Ordnung $\sqsubseteq'$ auf L^{2m} definiert man dabei

$$(l_1, \ldots, l_{2m}) \sqsubseteq' (l'_1, \ldots, l'_{2m}) \iff l_1 \sqsubseteq l'_1, \ldots, l_{2m} \sqsubseteq l'_{2m}.$$

Die Fixpunkte von G entsprechen dann genau den Lösungen des Gleichungssystems.

Satz 3.2.3 *Das oben angegebene Framework für das Beispiel aus Abschnitt 3.1 ist monoton. Dabei gilt* $\sqcup = \cap$.

Beweis:

[2] Transferfunktionen heißen manchmal auch Filterfunktionen (bei Bedingungs-Blöcken) bzw. Abstract Assignments (bei Zuweisungsblöcken). Diese Begriffen wurden so in der Vorlesung "Grundlagen der Softwarezuverlässigkeit" verwendet.

- Wir zeigen zunächst, dass jedes $f \in \mathcal{F}$ monoton ist. Sei $l \subseteq l'$, dann gilt $f(l) = l \backslash l_k \cup l_g \subseteq l' \backslash l_k \cup l_g = f(l')$.
- Außerdem enthält $\mathcal{F}$ die Identität, man muss nur $l_k = l_g = \emptyset$ setzen.
- Die Menge $\mathcal{F}$ sind auch abgeschlossen unter Funktionsverknüpfung. Seien $f, f' \in \mathcal{F}$ zwei Funktionen mit $f(l) = l \backslash l_k \cup l_g$ und $f'(l) = l \backslash l'_k \cup l'_g$. Dann gilt:

$$\begin{aligned}(f \circ f')(l) &= f(f'(l)) = f(l \backslash l'_k \cup l'_g) = (l \backslash l'_k \cup l'_g) \backslash l_k \cup l_g = (l \backslash l'_k) \backslash l_k \cup l'_g \backslash l_k \cup l_g \\ &= l \backslash (l'_k \cup l_k) \cup l'_g \backslash l_k \cup l_g.\end{aligned}$$

  Falls wir also $l''_k = l'_k \cup l_k$ und $l''_g = l'_g \backslash l_k \cup l_g$ setzen, so erhalten wir eine Funktion $f'' = f \circ f'$ mit $f''(l) = l \backslash l''_k \cup l''_g$, für die $f'' \in \mathcal{F}$ gilt.

□

Im folgenden werden wir einige weitere Beispiele für monotone Frameworks kennenlernen.

3.2.2 Analyse der Reichweite von Zuweisungen

Diese Analyse wird auch mit *Reaching Definitions Analysis* bezeichnet.

Wir wollen für jeden Block des Programms und für jede Variable alle Stellen bestimmen, an denen dieser Variable zuletzt ein Wert zugewiesen worden sein könnte. Dabei soll das Analyseergebnis eine Menge von Tupeln der Form $(\mathtt{x}, \ell)$ sein, was bedeutet, das die letzte Zuweisung an die Variable $\mathtt{x}$ im Block ℓ erfolgt sein könnte. Gibt es mehrere Blöcke, an denen der Variable $\mathtt{x}$ zuletzt ein Wert zugewiesen worden sein könnte, beispielsweise die Blöcke ℓ und ℓ', so muss das Analyseergebnis beide Paare $(\mathtt{x}, \ell)$ und $(\mathtt{x}, \ell')$ enthalten. Falls bisher noch keine Zuweisung an $\mathtt{x}$ stattgefunden haben könnte, so enthält die Menge das Element $(\mathtt{x}, ?)$.

Wir verwenden daher als Verband $L = \mathcal{P}(\mathbf{Var}_* \times (\mathbf{Lab}_* \cup \{?\}))$, wobei $\mathbf{Var}_*$ und $\mathbf{Lab}_*$ die in dem betrachteten Programm vorkommenden Variablen bzw. Labels sind.

Die Frage ist nun, wie die Ordnung auf den Verbandselementen aussehen soll? Verwenden wir wieder, wie bei der Analyse von verfügbaren Ausdrücken (Abschnitt 3.1) die Ordnung $\supseteq$ und damit $\cap$ als Supremumsoperation? Im jetzigen Fall wollen wir die Information über *alle* Pfade sammeln, die einen Block erreichen. Das heißt wir wollen wissen, ob es möglicherweise einen Pfad gibt, der diesen Block erreicht, wobei auf diesem Pfad der Variable $\mathtt{x}$ zuletzt im Block ℓ ein Wert zugewiesen wurde. In diesem Fall sollte das Analyseergebnis das Paar $(\mathtt{x}, \ell)$ enthalten. Es könnte jedoch auch andere Pfade geben, auf denen $\mathtt{x}$ in einem anderen Block zuletzt ein Wert zugewiesen wurde oder auf denen $\mathtt{x}$ noch gar kein Wert zugewiesen wurde. Informationen über diese Pfade sollten sich auch im Analyseergebnis widerspiegeln. Auf jeden Fall verwenden wir als Ordnung diesmal die übliche Mengeninklusion "$\subseteq$", womit wir dann $\cup$ als Supremum erhalten. In diesem Fall ist das Analyseergebnis auch umso ungenauer, je größer die betrachtete Menge ist.

Wir benutzen dieselbe Flussrelation wie im vorherigen Beispiel und setzen $E = \{\mathit{init}(S)\}$, $F = \mathit{flow}(S)$, wobei S das betrachtete Programm ist. D.h., es handelt sich auch hier um eine Vorwärts-Analyse. Der initiale Analysewert ist $\iota = \{(\mathtt{x}, ?) \mid \mathtt{x} \in \mathit{Var}(S)\}$. Dabei ist $\mathit{Var}(S)$ die Menge aller im Programm S vorkommenden Variablen.

Des weiteren verwenden wir dieselben Typen von Transferfunktionen, d.h., zu jeden Label ℓ gibt es Mengen $\mathit{kill}(\ell)$ und $\mathit{gen}(\ell)$, die den Elementen entsprechen, den gelöscht bzw. hinzugefügt werden sollen. In unserem jetzigen Beispiel sehen diese Funktionen folgendermaßen aus:

$$\begin{aligned}\mathit{kill}(\ell) &= \begin{cases} \{(\mathtt{x}, ?)\} \cup \{(\mathtt{x}, \ell') \mid \ell' \in \mathbf{Lab}_*\} & \text{falls } \mathit{block}(\ell) = (\mathtt{x}\mathtt{:=}a) \\ \emptyset & \text{sonst} \end{cases} \\ \mathit{gen}(\ell) &= \begin{cases} \{(\mathtt{x}, \ell)\} & \text{falls } \mathit{block}(\ell) = (\mathtt{x}\mathtt{:=}a) \\ \emptyset & \text{sonst} \end{cases}\end{aligned}$$

Und damit ergibt sich für die Transferfunktionen:

$$f_\ell(l) = l \backslash kill(\ell) \cup gen(\ell),$$

wobei $l \in \mathcal{P}(\mathbf{Var}_* \times (\mathbf{Lab}_* \cup \{?\}))$.

Eine mögliche Anwendung der Analyse der Reichweite von Zuweisungen ist die Zuordnung von Blöcken, die einer Variable einen Wert zuweisen, zu Blöcken, die diese Variable benutzen. Hier gibt es Optimierungsmöglichkeiten, z.B. die Eliminierung von totem Code, die erfolgen kann, wenn einer Variable `x` in einem Block ℓ ein Wert zugewiesen wird, das Tupel $(\mathtt{x}, \ell)$ aber niemals in dem Analyseergebnis eines Blocks auftaucht, in dem diese Variable `x` benutzt wird. In diesem Fall ist der Block ℓ überflüssig und kann entfernt werden.

Aufgabe 3.2.4 Betrachten Sie folgendes Programmstück:

$$[\mathtt{x:=0}]^1;[\mathtt{x:=3}]^2;\mathtt{if}\ [\mathtt{x=y}]^3\ \mathtt{then}\ [\mathtt{y:=3}]^4\ \mathtt{else}\ [\mathtt{y:=5}]^5\ \mathtt{fi};[\mathtt{y:=x}]^6$$

Führen Sie eine Analyse der Reichweite von Zuweisungen aus und argumentieren Sie mit Hilfe dieses Analyse-Ergebnisses, welche Anweisungen des Programms toter Code sind und daher entfernt werden können.

Allgemein kann man mit dieser Methode toten Code (oder zumindest einen Teil des toten Codes) folgendermaßen identifizieren: falls $[\mathtt{y:=b}]^{\ell'}$ in dem Programm vorkommt und für jeden Block der Form $[\mathtt{x:=}a]^\ell$ mit $\mathtt{y} \in \mathbf{AExp}(a)$ und für jeden Block der Form $[\mathtt{b}]^\ell$ mit $\mathtt{y} \in \mathbf{AExp}(b)$ gilt: $(\mathtt{y}, \ell') \notin \mathsf{A}_\circ(\ell)$, dann kann $[\mathtt{y:=b}]^{\ell'}$ entfernt werden, ohne dass sich an dem Programmablauf etwas ändert. In diesem Fall wird `y` nämlich definiert, aber anschließend nicht verwendet. Falls die Ausgabe des Programms in `y` stehen kann, dann muss natürlich noch darauf geachtet werden, dass $(\mathtt{y}, \ell')$ nicht in $\mathsf{A}_\circ(\ell)$ für finale Blöcke ℓ vorkommt.

Aufgabe 3.2.5 Führen Sie eine Analyse der Reichweite von Zuweisungen auf folgendem Programm durch, das die Fakultät von dem in Variable `x` gespeicherten Wert bestimmt:

```
[y := x]^1;
[z := 1]^2;
while [y>1]^3 do
   [z := z*y]^4;
   [y := y-1]^5
od;
[y:=0]^6
```

3.2.3 Analyse lebendiger Variablen

In [NNH99] wird diese Analyse als *Live Variables Analysis* bezeichnet.

Eine Variable heißt *lebendig* am Ausgang eines Blocks, wenn es einen Pfad von diesem Block zu einem anderen Block gibt, der diese Variable in einer Bedingung oder auf der rechten Seite einer Zuweisung benutzt, ohne dass die Variable vorher neu definiert wird. Die Analyse soll zu jedem Block bestimmen, welche Variablen am Ausgang dieses Blocks möglicherweise lebendig sind. Diese Information kann einem Compiler wiederum zur Optimierung dienen. Falls ein Block die Form $[\mathtt{x:=}a]^\ell$ hat, die Variable `x` am Ausgang dieses Blocks jedoch nicht lebendig ist, so kann dieser Block entfernt werden (ähnlich wie in Abschnitt 3.2.2).

Der Verband L ist hier relativ einfach zu bestimmen: wir verwenden einfach die Menge aller Mengen von im Programm vorkommenden Variablen, d.h., $L = \mathcal{P}(\mathbf{Var}_*)$. Da es reicht, dass die jeweilige Variable in *einem* Pfad verwendet wird, vereinigen wir die Information über alle Pfade und verwenden $\subseteq$ als Ordnung und damit $\cup$ als Supremumsoperation.

Eines ist jedoch anders: wir wollen in diesem Fall Aussagen über die Zukunft eines Blocks ℓ machen, dazu müssen wir rückwärts alle Pfade von den Endzuständen zum Block ℓ verfolgen und Informationen darüber aufsammeln, welche Variablen verwendet werden. Wir setzen daher

$E = \mathit{final}(S)$ und $F = \mathit{flow}^R(S)$, wobei $\mathit{flow}^R(S) = \{(\ell', \ell) \mid (\ell, \ell') \in \mathit{flow}(S)\}$. Damit handelt es sich um eine Rückwärtsanalyse. Des weiteren setzen wir ι, den initialen Analysewert, auf eine vorher festgelegt Menge von Variablen, die die Ausgabe des Programms darstellen sollen. Genau diese werden am Ende des Programms noch benötigt.

Die Transferfunktionen können wir wieder ähnlich wie in den vorhergehenden Beispielen definieren.

$$\mathit{kill}(\ell) = \begin{cases} \{\mathtt{x}\} & \text{falls } \mathit{block}(\ell) = (\mathtt{x:=}a) \\ \emptyset & \text{sonst} \end{cases}$$

$$\mathit{gen}(\ell) = \begin{cases} \mathit{Var}(a) & \text{falls } \mathit{block}(\ell) = (\mathtt{x:=}a) \\ \mathit{Var}(b) & \text{falls } \mathit{block}(\ell) = b \\ \emptyset & \text{sonst} \end{cases}$$

Dann kann man die Transferfunktionen wie in Abschnitt 3.2.2 definieren. Zu beachten ist dabei allerdings, dass der Analysewert, der $\mathsf{A}_\circ$ entspricht, in Datenflussrichtung dem *Ausgang* eines Blocks zuzuordnen ist. Ebenso entspricht jetzt $\mathsf{A}_\bullet$ dem *Eingang* eines Blocks.

Aufgabe 3.2.6 Führen Sie bei folgendem Programmstück eine Analyse lebendiger Variablen durch. Wir legen dabei $\iota = \{\mathtt{x}\}$ fest.

```
[x:=2]1;
[y:=4]2;
[x:=1]3;
if [y>x]4
   then [z:=2*x]5
   else [z:=y*y]6
fi;
[x:=z]7
```

3.2.4 Analyse benötigter Ausdrücke

Diese Analyse wird im Englischen auch *Very Busy Expressions Analysis* genannt. Wir wollen diejenigen arithmetischen Ausdrücke bestimmen, die auf jedem Pfad von einem bestimmten Block aus auf jeden Fall benutzt werden, ohne vorher verändert zu werden. Beispielsweise wird in dem Programmstück

$$\mathtt{if}\ [\mathtt{x=1}]^1\ \mathtt{then}\ [\mathtt{y:=a+b*c}]^2\ \mathtt{else}\ [\mathtt{y:=b*c}]^3\ \mathtt{fi}$$

der arithmetische Ausdruck `b*c` auf jedem Pfad benutzt, der vom Block 1 ausgeht. Eine mögliche Optimierung ist, diesen Ausdruck bereits in Block 1 zu berechnen und dann jeweils in den Blöcken 2 und 3 zu verwenden.

Als Verband benutzen wir , wie in dem in Abschnitt 3.1 vorgestellten Beispiel den Potenzmengenverband $\mathcal{P}(\mathbf{AExp}_*)$, wobei $\mathbf{AExp}_*$ die Menge der in dem zu analysierenden Programmstück vorkommenden arithmetischen Ausdrücke ist. Da wir nur arithmetische Ausdrücke in Analyseergebnis aufnehmen wollen, die in *allen* Pfaden verwendet werden, benötigen wir den Schnitt als Supremumsoperation und verwenden daher $\supseteq$ als Ordnung.

Des weiteren handelt es sich um eine Rückwärtsanalyse, daher setzen wir, wie in Abschnitt 3.2.3 $E = \mathit{final}(S)$ und $F = \mathit{flow}^R(S)$. Der initiale Analysewert ist $\iota = \emptyset$.

Nun müssen wir nur noch die in jedem Block zu löschenden und hinzuzufügenden arithmetischen Ausdrücke angeben:

$$\mathit{kill}(\ell) = \begin{cases} \{a' \in \mathbf{AExp}_* \mid \mathtt{x} \in \mathit{Var}(a')\} & \text{falls } \mathit{block}(\ell) = (\mathtt{x:=}a) \\ \emptyset & \text{sonst} \end{cases}$$

$$\mathit{gen}(\ell) = \begin{cases} \mathbf{AExp}(a) & \text{falls } \mathit{block}(\ell) = (\mathtt{x:=}a) \\ \mathbf{AExp}(b) & \text{falls } \mathit{block}(\ell) = b \\ \emptyset & \text{sonst} \end{cases}$$

Aufgabe 3.2.7 Führen Sie bei folgendem Programmstück eine Analyse benötigter Ausdrücke durch:

```
if [a>b]^1
   then [x:=b-a]^2;[y:=a-b]^3
   else [y:=b-a]^4;[x:=a-b]^5
fi
```

3.2.5 Zusammenfassung: Monotone Frameworks

In den bisherigen Beispielen haben wir als Verband immer einen Potenzmengenverband benutzt. Es gibt jedoch auch Beispiele, bei denen eine andere Verbandsstruktur günstiger ist. Solche Verbände kommen in Aufgabe 3.2.8 vor, ein weiteres Beispiel ist die Datenflussanalyse beim Java Bytecode Verifier, der in Kapitel 4 vorgestellt wird.

Verwendet man jedoch einen Potenzmengenverband, so kann man sich leicht folgendes als Daumenregel überlegen: wenn es ausreicht, dass das gesuchte Ereignis in *einem* Pfad eintritt, so verwendet man $\cup$ als Supremumsoperator und verwendet damit $\subseteq$ als Ordnung. Falls das Ereignis in *allen* Pfaden eintreten soll, so verwendet man dagegen $\cap$ als Supremum und $\supseteq$ als Ordnung.

Redet man über alle Pfade in der Vergangenheit eines Blocks, so macht man eine Vorwärtsanalyse und verwendet $\mathit{flow}(S)$ als Flussrelation. Spricht man über die Zukunft eines Blocks, so handelt es sich um eine Rückwärtsanalyse und $\mathit{flow}^R(S)$ wird als Flussrelation verwendet.

Die vier Beispiele, die bisher behandelt wurden, sind in Tabelle 3.3 zusammengefasst.

	Verfügbare Ausdrücke	Reichweite von Zuweisungen	Benötigte Ausdrücke	Lebendige Variable
L	$\mathcal{P}(\mathbf{AExp}_*)$	$\mathcal{P}(\mathbf{Var}_* \times (\mathbf{Lab}_* \cup \{?\}))$	$\mathcal{P}(\mathbf{AExp}_*)$	$\mathcal{P}(\mathbf{Var}_*)$
$\sqsubseteq$	$\supseteq$	$\subseteq$	$\supseteq$	$\subseteq$
$\sqcup$	$\cap$	$\cup$	$\cap$	$\cup$
$\bot$	$\mathbf{AExp}_*$	$\emptyset$	$\mathbf{AExp}_*$	$\emptyset$
ι	$\emptyset$	$\{(\mathtt{x}, ?) \mid \mathtt{x} \in \mathit{Var}(S)\}$	$\emptyset$	variabel
E	$\{\mathit{init}(S)\}$	$\{\mathit{init}(S)\}$	$\mathit{final}(S)$	$\mathit{final}(S)$
F	$\mathit{flow}(S)$	$\mathit{flow}(S)$	$\mathit{flow}^R(S)$	$\mathit{flow}^R(S)$
$\mathcal{F}$	$\{f\colon L \to L \mid f \text{ monoton}\}$			
f_ℓ	$f_\ell(A) = A \backslash \mathit{kill}(\ell) \cup \mathit{gen}(\ell)$			

Tabelle 3.3: Instanzen von monotonene Frameworks

Aufgabe 3.2.8 Bestimmen Sie zu den folgenden Analysen jeweils das benötigte monotonen Framework. Zunächst sollten Sie festlegen, wie der verwendete Verband aussehen soll, wie die Ordnung festgelegt sein sollte und ob es sich um eine Vorwärts- oder Rückwärts-Analyse handelt. Anschließend sollten Sie den initialen Analysewert und die Transferfunktionen bestimmen.

(a) Bestimmen Sie zu jedem Block die Menge der Variablen, die am Eingang dieses Blockes auf jeden Fall noch uninitialisiert sind.

(b) Bestimmen Sie zu jedem Block die Menge der Variablen, die am Eingang dieses Blockes möglicherweise noch uninitialisiert sind, d.h., denen noch kein ein Wert zugewiesen wurde. Dabei kann eine bereits initialisierte Variable auch wieder zu einer uninitialisierten Variable werden, sobald ihr ein Ausdruck zugewiesen wird, der eine aktuell uninitialisierte Variable enthält.

(c) Bestimmen Sie zu jedem Block die Menge der (Integer-)Konstanten, die möglicherweise im weiteren Verlauf des Programms noch benutzt werden. Wir sagen, eine Konstante c wird benutzt, wenn es einen Block $[\mathtt{x:=}a]^\ell$ gibt, so dass c in dem arithmetischen Ausdruck a vorkommt und dieser Block ℓ durchlaufen wird.

(d) Bestimmen Sie zu jedem Block eine Funktion $\mathbf{Var}_* \to \mathbf{Lab}_* \cup \{\mathit{undef}, \mathit{conflict}\}$, wobei einer Variable $\mathtt{x}$ der Wert *undef* zugeordnet wird, falls die Variable auf jedem Pfad zu diesem Block mit Sicherheit nicht definiert wurde. Ein Label ℓ wird zugeordnet, wenn die Variable entweder noch nicht definiert wurde, oder es höchstens einen Block ℓ gibt, in dem sie zuletzt definiert wurde. Dahingegen wird der Wert *conflict* verwendet, wenn es mehrere Blöcke geben könnte, an denen die Variable zuletzt definiert wurde.

Machen Sie sich insbesondere darüber Gedanken, wie der zugrundeliegende Verband L aussehen aussehen soll, ein Potenzmengenverband ist hier nicht so günstig.

3.3 Lösen von Fixpunkt-Gleichungen

Um die kleinste Lösung der Gleichungen für $\mathsf{A}_\circ$ und $\mathsf{A}_\bullet$ aus Abschnitt 3.2.1 zu bestimmen, können wir eine klassische Fixpunkt-Iteration anwenden, wie es in Abschnitt 3.1 demonstriert wurde. Da dies jedoch recht aufwendig zu implementieren ist und viele Werte mehrfach berechnet werden, verwendet man im allgemeinen den folgenden sogenannten Worklist-Algorithmus. Die Ausgabe dieses Algorithmus bezeichnen wir mit $\mathit{MFP}_\circ$ und $\mathit{MFP}_\bullet$, wobei *MFP* für *minimal fixed-point* steht (manchmal auch *maximal fixed-point*). Als Hilfsvariablen werden dabei eine Menge W (die Worklist) benutzt, die den noch abzuarbeitenden Teil der Flussrelation F enthält, und ein Array A, so dass $\mathsf{A}[\ell]$ den bisher berechneten Analysewert für den Eingang zu Block ℓ enthält.

Algorithmus 3.3.1 (Worklist-Algorithmus)

Eingabe: eine Instanz eines monotonen Frameworks: $(L, \mathcal{F}, F, \mathbf{Lab}, E, \iota, f.)$

Ausgabe: $\mathit{MFP}_\circ$, $\mathit{MFP}_\bullet$

Schritt 1 (Initialisierung)

```
W := F;                          (die Flussrelation in die Worklist aufnehmen)
for all ℓ ∈ Lab do               (Anfangswerte setzen)
    if ℓ ∈ E                     (Extremale Labels erhalten den Wert ι)
        then A[ℓ] := ι
        else A[ℓ] := ⊥
    fi
od
```

Schritt 2 (Iteration)

```
while W ≠ ∅ do
    choose (ℓ', ℓ) from W;       (Ein Paar aus der Worklist nehmen)
    W := W\{(ℓ', ℓ)};
    if f_ℓ'(A[ℓ']) ⋢ A[ℓ] then   (Ändert sich dadurch der Analysewert von ℓ?)
        A[ℓ] := A[ℓ] ⊔ f_ℓ'(A[ℓ']);   (Falls ja: Analysewert von ℓ korrigieren)
        for all ℓ'' with (ℓ, ℓ'') ∈ F do   (Alle Nachfolger von ℓ in die Worklist aufnehmen)
            W := W ∪{(ℓ, ℓ'')}
        od
    fi
od
```

Schritt 3 (Ausgabe)

```
for all ℓ ∈ Lab do               (Analysewerte für Block-Ein- und Ausgänge berechnen)
```

$MFP_\circ(\ell) := \mathsf{A}[\ell]$;
$MFP_\bullet(\ell) := f_\ell(\mathsf{A}[\ell])$
od

Es ist jetzt zu zeigen, dass dieser Algorithmus tatsächlich terminiert und dass das Ergebnis die kleinste Lösung des Gleichungssystems ist.

Satz 3.3.2 (Terminierung des Worklist-Algorithmus) *Wir nehmen an, dass L ein Verband ist, in dem die Ascending Chain Condition gilt. Dann terminiert der Worklist-Algorithmus.*

Beweis: Wir nehmen an, dass jeder Block maximal b Nachfolger hat.

Die Schritte 1 und 3 terminieren trivialerweise. In Schritt 2 wird in jedem Durchgang ein Paar aus W gelöscht und es werden maximal b neue Paare zur Worklist hinzugefügt. Dies ist jedoch nur dann der Fall, wenn $\mathsf{A}[\ell]$ für mindestens ein $\ell \in \mathbf{Lab}$ größer wird. Da der Verband allerdings keine unendlich langen ansteigenden Ketten enthält, kann dies nur endlich oft passieren und die Worklist ist irgendwann leer. □

Wenn h die Länge der längsten Kette in L ist, dann hat der Algorithmus die Laufzeit $O(|F| \cdot h)$, da nach dem Terminierungsbeweis die Worklist höchstens h-mal mit F aufgefüllt wird.

Satz 3.3.3 (Korrektheit des Worklist-Algorithmus) *Die Ausgabe des Worklist-Algorithmus ist die kleinste Lösung der Gleichungen* (3.1) *und* (3.2) *aus Abschnitt 3.2.1.*

Beweis: Wir nehmen an, dass $\mathsf{A}_\circ(\ell)$, $\mathsf{A}_\bullet(\ell)$ die kleinste Lösung des Gleichungssystems darstellen.

Wir zeigen nun die Korrekheit des Algorithmus in mehreren Schritten.

(a) Wir beweisen zunächst folgende Invariante: zu jedem Zeitpunkt gilt $\mathsf{A}[\ell] \sqsubseteq \mathsf{A}_\circ(\ell)$.

Dies gilt offensichtlich nach der Initialisierung von A, denn entweder ist $\mathsf{A}[\ell] = \bot_L$, falls $\ell \notin E$, oder $\mathsf{A}[\ell] = \iota \sqsubseteq \mathsf{A}_\circ(\ell)$, falls $\ell \in E$.

Bei der Iteration gibt es nur eine Zuweisung der Form $\mathsf{A}[\ell] := \mathsf{A}[\ell] \sqcup f_{\ell'}(\mathsf{A}[\ell'])$ an A, wobei $(\ell', \ell) \in F$. Wir bezeichnen mit ${}'\mathsf{A}$ bzw. A' den Array A vor und nach der Zuweisung. Es gilt:

$$\begin{array}{lcll} \mathsf{A}'[\ell] & = & {}'\mathsf{A}[\ell] \sqcup f_{\ell'}({}'\mathsf{A}[\ell']) & \\ & \sqsubseteq & \mathsf{A}_\circ(\ell) \sqcup f_{\ell'}(\mathsf{A}_\circ(\ell')) & \text{(Monotonie)} \\ & = & \mathsf{A}_\circ(\ell) \sqcup \mathsf{A}_\bullet(\ell') & \text{(Gleichung (3.2))} \\ & = & \mathsf{A}_\circ(\ell) & \text{(Gleichung (3.1))} \end{array}$$

(b) Wir zeigen durch einen Widerspruchsbeweis, dass nach Terminierung des Algorithmus für jedes $(\ell', \ell) \in F$ gilt: $f_{\ell'}(\mathsf{A}[\ell']) \sqsubseteq \mathsf{A}[\ell]$. Angenommen, dem wäre nicht so, dann gibt es ein $(\ell', \ell) \in F$ mit $f_{\ell'}(\mathsf{A}[\ell']) \not\sqsubseteq \mathsf{A}[\ell]$.

Wir betrachten die Stelle, an der $\mathsf{A}[\ell']$ das letzte Mal ein Wert zugewiesen wurde. Falls dies in Schritt 1 der Fall war, so wurde (ℓ', ℓ) in W aufgenommen und es wurde in Schritt 2 sichergestellt, dass $f_{\ell'}(\mathsf{A}[\ell']) \sqsubseteq \mathsf{A}[\ell]$ gilt. Da $\mathsf{A}[\ell']$ anschließend nicht mehr geändert wird und $\mathsf{A}[\ell]$ nur wachsen kann, ist dies auch bei Terminierung des Algorithmus der Fall, was ein Widerspruch ist.

Falls $\mathsf{A}[\ell']$ zum letzten Mal in Schritt 2 verändert wurde, so wurde ebenfalls (ℓ', ℓ) in W aufgenommen und es gilt dieselbe Argumentation wie im vorherigen Fall. Daher ergibt sich auch hier ein Widerspruch.

(c) Aus der Argumentation in (b) und $\iota \sqsubseteq \mathsf{A}[\ell]$ falls $\ell \in E$ (siehe Initialisierungs-Phase) ergibt sich damit:

$$\bigsqcup \{f_{\ell'}(\mathsf{A}[\ell']) \mid (\ell', \ell) \in F\} \sqcup \iota^\ell_E \sqsubseteq \mathsf{A}[\ell]$$

bei Terminierung des Algorithmus. D.h., A ist ein Präfixpunkt.

Da $\mathsf{A}_\circ$ der kleinste Fixpunkt, aber auch der kleinste Präfixpunkt im Sinne obiger Ungleichung ist, ergibt sich damit bei Terminierung:

$$\forall\, \ell \in \mathbf{Lab}\colon \mathsf{A}_\circ(\ell) \sqsubseteq \mathsf{A}[\ell] = \mathit{MFP}_\circ(\ell).$$

Und zusammen mit der Invariante aus Punkt (a) erhält man damit:

$$\forall\, \ell \in \mathbf{Lab}\colon \mathsf{A}_\circ(\ell) = \mathit{MFP}_\circ(\ell).$$

Mit Hilfe von Gleichung (3.2) ergibt sich dann unmittelbar:

$$\forall\, \ell \in \mathbf{Lab}\colon \mathsf{A}_\bullet(\ell) = \mathit{MFP}_\bullet(\ell).$$

□

3.4 Optimierung im Gnu C Compiler

Um zu sehen, wie Compiler-Optimierung in der Praxis funktioniert, betrachten wir zwei kleine Programme und überprüfen, wie der Compiler sie mit und ohne Optimierungen in x86-Assemblercode (einer Vorstufe zum eigentlichen Maschinencode) übersetzt. Andere Beispiele findet man in [Nar].

Der Gnu C Compiler wird dabei mit folgenden Optionen aufgerufen:

`gcc -S program.c -o program.s`	Kompilieren in Assemblercode ohne Optimierung
`gcc -S -O3 program.c -o program.s`	Kompilieren in Assemblercode mit Optimierung

Der Assemblercode ist allerdings nicht ganz einfach zu verstehen, er wird im folgenden kommentiert.

Wir betrachten zunächst folgendes kleine C-Programm:

```
int f() {
        int a;

        a = 1;
        a = 2;

        return a;
}
```

Man erhält folgende Ausgabe, zunächst der nicht-optimierte Assemblercode:

```
        .file   "live-var.c"        ; zunaechst einige Definitionen
        .version        "01.01"
gcc2_compiled.:
.text
        .align 4
.globl f
        .type    f,@function
f:
        pushl   %ebp                ; %epb, der alte Stack Base Pointer wird
                                    ; auf den Stack gepusht und damit gesichert
        movl    %esp, %ebp          ; Der jetzige Stack Pointer wird
                                    ; der Base Pointer
        subl    $4, %esp            ; Subtrahiere 4 vom Stack Pointer
                                    ; -> 4 Bytes werden auf dem Stack reserviert
        movl    $1, -4(%ebp)        ; Lege die 1 auf den Stack  an die Stelle
```

```
                                ; Base Pointer - 4
        movl    $2, -4(%ebp)    ; Lege die 2 auf den Stack an die Stelle
                                ; Base Pointer - 4
        movl    -4(%ebp), %eax  ; Lege das oberste Element des Stacks
                                ; in das Register %eax
        movl    %eax, %eax      ; Rueckgabewert in Register %eax
        leave                   ; Aus Funktionsaufruf zurueckkehren
        ret
.Lfe1:
        .size    f,.Lfe1-f
        .ident  "GCC: (GNU) 2.96 20000731 (Red Hat Linux 7.3 2.96-110)"
```

Optimiert erhält man folgenden Code:

```
        .file   "live-var.c"    ; Wieder einige Definitionen
        .version        "01.01"
gcc2_compiled.:
.text
        .align 4
.globl f
        .type    f,@function
f:
        pushl   %ebp            ; Base Pointer sichern
        movl    %esp, %ebp      ; und aktueller Stack Pointer wird
                                ; Base Pointer
        movl    $2, %eax        ; Zahl 2 in Register %eax
        popl    %ebp            ; Base Pointer wieder vom Stack nehmen
        ret                     ; Aus Funktionsaufruf zurueckkehren
.Lfe1:
        .size    f,.Lfe1-f
        .ident  "GCC: (GNU) 2.96 20000731 (Red Hat Linux 7.3 2.96-110)"
```

Man sieht dabei deutlich, dass der Code zur ersten Zuweisung (`a:=1`) verschwunden ist. Das kann beispielsweise mit der Analyse lebendiger Variablen erreicht werden, denn die Variable `a` ist nach der ersten Zuweisung nicht lebendig. Zusätzlich werden noch einige Stack-Operationen eingespart und der Rückgabewert direkt ins Register `%eax` geschrieben.

Das nächste Beispiel ist folgender C-Code:

```
int f(int a,int b) {
        int c;

        c = a*b;

        if (c > 0)
          {return a*b+1;}
        else
          {return a*b+2;}
}
```

Nicht-optimiert ergibt sich folgender Assemblercode. Um den Code zu verstehen, ist es nützlich zu wissen, dass bei Eintritt in eine Prozedur die Rücksprungadresse zuoberst auf dem Stack liegt. Darauf folgenden die Parameter, mit denen die Prozedur aufgerufen wurde.

```
        .file   "avail-expr2.c"
        .version        "01.01"
```

```
gcc2_compiled.:
.text
        .align 4
.globl f
        .type    f,@function
f:
        pushl   %ebp                ; %epb, der alte Stack Base Pointer wird
                                    ; auf den Stack gepusht und damit gesichert
        movl    %esp, %ebp          ; Der jetzige Stack Pointer wird
                                    ; der Base Pointer
        subl    $4, %esp            ; 4 Byte Platz auf dem Stack fuer c anlegen
        movl    8(%ebp), %eax       ; Wert Base Pointer + 8 (= b) in Register %eax
        imull   12(%ebp), %eax      ; Wert Base Pointer + 12 (= a) zu Register %eax
                                    ; dazumultiplizieren
        movl    %eax, -4(%ebp)      ; Ergebnis der Multiplikation auf Stack
                                    ; in Variable c
        cmpl    $0, -4(%ebp)        ; Vergleiche 0 mit c
        jle     .L3                 ; Falls 0 kleiner gleich c -> Springe nach .L3
        movl    8(%ebp), %eax       ; Wert Base Pointer + 8 (= b) in Register %eax
        imull   12(%ebp), %eax      ; Wert Base Pointer + 12 (= a) zu Register %eax
                                    ; dazumultiplizieren
        incl    %eax                ; Inkrementiere %eax um 1
        movl    %eax, %eax          ; Rueckgabewert in Register %eax
        jmp     .L2                 ; Sprung nach .L2
        .p2align 2
.L3:
        movl    8(%ebp), %eax       ; Wert Base Pointer + 8 (= b) in Register %eax
        imull   12(%ebp), %eax      ; Wert Base Pointer + 12 (= a) zu Register %eax
                                    ; dazumultiplizieren
        addl    $2, %eax            ; Zahl 2 zu Register %eax addieren
        movl    %eax, %eax          ; Rueckgabewert in Register %eax
.L2:
        leave                       ; Rueckkehr aus Funktionsaufruf
        ret
.Lfe1:
        .size    f,.Lfe1-f
        .ident  "GCC: (GNU) 2.96 20000731 (Red Hat Linux 7.3 2.96-110)"
```

Und jetzt der optimierte Code:

```
        .file   "avail-expr2.c"     ; zunaechst einige Definitionen
        .version        "01.01"
gcc2_compiled.:
.text
        .align 4
.globl f
        .type    f,@function
f:
        pushl   %ebp                ; %epb, der alte Stack Base Pointer wird
                                    ; auf den Stack gepusht und damit gesichert
        movl    %esp, %ebp          ; Der jetzige Stack Pointer wird
                                    ; der Base Pointer
        movl    12(%ebp), %eax      ; Wert Base Pointer + 12 (= a) in Register %eax
        imull   8(%ebp), %eax       ; Wert Base Pointer + 8 (= b) zu Register %eax
```

```
                                    ; dazumultiplizieren
        testl   %eax, %eax          ; Vergleiche (%eax AND %eax) mit 0
        jle     .L3                 ; Falls kleiner gleich -> nach .L3 springen
        incl    %eax                ; %eax um 1 inkrementieren
        jmp     .L5                 ; nach .L5 springen
        .p2align 2
.L3:
        addl    $2, %eax            ; Zahl 2 zu %eax addieren
.L5:
        popl    %ebp                ; Base Pointer wieder vom Stack nehmen
        ret                         ; Aus Funktionsaufruf zurueckkehren
.Lfe1:
        .size    f,.Lfe1-f
        .ident  "GCC: (GNU) 2.96 20000731 (Red Hat Linux 7.3 2.96-110)"
```

Hier verschwindet also die Multiplikation von `a` und `b` in den beiden Ästen der if-then-else-Anweisung, da der entsprechende Ausdruck vorher schon berechnet wurde. Das kann beispielsweise mit der Analyse verfügbarer Ausdrücke erkannt werden. Des weiteren wird in der optimierten Version kein Platz für `c` auf dem Stack angelegt, sondern die entsprechenden Werte in den Registern gehalten.

Kapitel 4

Java Bytecode Verifier

Eines der Merkmale der Programmiersprache Java ist der Einsatz von Bytecode, einer Art von "maschinenunabhängigem Maschinencode". Java-Programme werden in den Bytecode übersetzt, der dann auf andere Rechner übertragen und dort ausgeführt wird, indem er von der Java Virtual Machine (JVM) interpretiert wird. Die Übersetzung in Bytecode ist ein Kompromiss zwischen klassischem Kompilieren, bei dem das Programm in die jeweilige Maschinensprache übersetzt wird, damit sehr effizient ausgeführt werden kann, aber maschinenabhängig wird, und Interpretation, bei der direkt der Code einer Hochsprache ausgeführt wird.

Da Bytecode oft über das Netz von unbekannten Servern geholt wird, ist es ganz besonders wichtig, vor der Ausführung zu überprüfen, ob der Code korrekt ist und keine Laufzeitfehler verursacht. Da Java eine objekt-orientierte Sprache ist und sich dies auch im Bytecode wiederspiegelt, ist besonders darauf zu achten, dass bei Methodenaufrufen eines Objekts einer bestimmten Klasse, diese Klasse auch wirklich die entsprechenden Methoden besitzt und die erwarteten Rückgabewerte zurückgibt. Außerdem muss überprüft werden, dass keine Integer-Werte an Stelle von Referenzen auf Objekte verwendet werden. Dies könnte dazu führen, dass ein Java-Applet auf beliebige Stellen des Hauptspeichers zugreifen könnte.

Diese Analyse-Aufgabe übernimmt der sogenannte Java Bytecode Verifier, der im wesentlichen eine Instanz eines monotonen Frameworks ist. Es gibt jedoch auch Aspekte, die eher dem Bereich der Typsysteme zuzuordnen sind.

Ein weiterer Effekt des Java Bytecode Verifiers ist es, dass – dadurch, dass die Höhe des Stacks an jeder Programmstelle eindeutig ist – jeder Stackoperation eindeutige Stack-Adressen zugeordnet werden können. Das vereinfach die Adressarithmetik und führt zu schnellerer Ausführung.

Der Java Bytecode Verifier wird in der Java Virtual Machine Specification [LY99] vorgestellt. Eine formalere Beschreibung findet sich beispielsweise in [Nip01, KN01, Kle03].

4.1 Befehle der Java Virtual Machine

Wir beschränken uns hier auf die Beschreibung der Analyse einer Methode. Zu jeder Methode gehört ein Satz von m lokalen Registern und ein Stack, der höchstens die Länge max haben kann. An Datentypen betrachten wir Integer, NullT (Null Pointer) und Class *cname* (Referenz auf ein Objekt der Klasse *cname*). Die letzten beiden Datentypen bezeichnen Referenzen, sie erfüllen das Prädikat *isref*. Die Menge aller Datentypen bezeichnen wir mit Ty.

Ein typischer Registersatz und ein Stack einer Methode könnten beispielsweise aussehen wie in Abbildung 4.1. Auch die Typen die einzelnen Einträge sind angegeben.

Wir betrachten den folgenden eingeschränkten aber repräsentativen Satz von Befehlen. Dabei nehmen wir vereinfachend an, dass jede Methode mit genau einem Parameter aufgerufen wird. Eine Methode besteht aus einer mit Zeilennummern von 1 bis k versehenen Liste von Befehlen. Des weiteren nehmen wir an, dass die betrachtete Methode einen Rückgabewert vom Typ ty_r haben soll. Die Befehle lauten wie folgt:

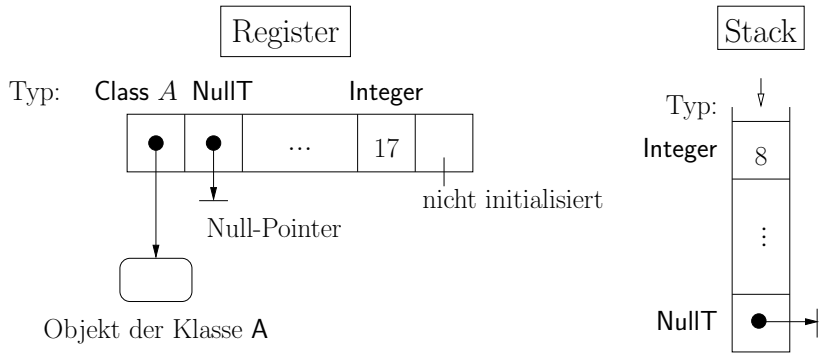


Abbildung 4.1: Beispiel für Registersatz und Stack

- Load n: Lege den Inhalt des n-ten Registers auf den Stack (Push-Operation)
- Store n: Entferne das oberste Element vom Stack und weise es dem n-ten Register zu (Pop-Operation)
- AConst_Null: Lege einen Null-Pointer zuoberst auf den Stack (Push-Operation)
- IConst i: Lege den Integer-Wert i zuoberst auf den Stack (Push-Operation)
- IAdd: Entferne die beiden obersten Werte vom Stack, addiere sie und lege das Ergebnis wieder auf den Stack. Dabei wird angenommen, dass es sich bei diesen beiden Werten um Integer-Werte handelt.
- Getfield *fname ty cname*: Hier wird angenommen, dass das zuoberst auf dem Stack liegende Element eine Referenz auf ein Objekt der Klasse *cname* ist. Diese Referenz wird vom Stack entfernt und der Inhalt des Feldes *fname* vom Typ *ty* dieses Objekts stattdessen auf den Stack gelegt.
- Putfield *fname ty cname*: Zunächst werden die obersten beiden Elemente vom Stack entfernt, wobei angenommen wird, dass das zweite Element eine Referenz auf ein Objekt der Klasse *cname* darstellt. Dem Feld *fname* vom Typ *ty* dieses Objekts wird dann das erste Element zugewiesen.
- New *cname*: Ein neues Objekt der Klasse *cname* wird erzeugt und auf den Stack gelegt (Push-Operation).
- Invoke *cname mname* ty_1 ty_2: Die obersten beiden Elemente werden vom Stack entfernt, wobei das erste der zu übergebende Parameter vom Typ ty_1 ist. Das zweite Element des Stacks soll eine Referenz auf ein Objekt der Klasse *cname* darstellen, dessen Methode *mname* aufgerufen wird. Der Rückgabewert soll vom Typ ty_2 sein und wird nach Rückkehr aus der aufgerufenen Methode auf den Stack gelegt.
- CmpEq q: Die obersten beiden Werte werden vom Stack genommen und miteinander verglichen. Dabei sollten diese Werte entweder beide vom Typ Integer oder beide Referenzen sein. Bei Gleichheit springe in Zeile q.
- Return: Kehre an die Aufrufstelle der Methode zurück und gibt den zuoberst auf dem Stack liegenden Wert zurück. Dieser sollte vom Typ ty_r sein.

Bei Eintritt in eine Methode ist der Stack leer, das erste Register enthält eine Referenz auf das Objekt, dessen Methode aufgerufen wird, und das zweite Register enthält den Parameter der Funktion. Falls die Funktion weitere Parameter hat, befinden sich diese in den weiteren Registern,

wir werden jedoch zur Vereinfachung nur den Fall mit einem Parameter betrachten. Alle anderen Register sind uninitialisiert.

In den obenstehenden Befehlsbeschreibungen kommen Formulierungen der Form "das zuoberst auf dem Stack liegende Element sollte vom Typ *ty*" sein. Des weiteren wird bei vielen Operationen vorausgesetzt, dass der Stack nicht leer ist, sondern mindestens ein oder zwei Elemente enthält. Sind diese Bedingungen nicht erfüllt, so ergäbe sich bei Ausführung des Codes ein Laufzeitfehler. Die Aufgabe des nun vorgestellten Analyseverfahrens ist es, zu vermeiden, dass solche Laufzeitfehler entstehen können.

Es folgt eine (unvollständige) Auflistung von Fehlern, die vom Java Bytecode Verifier erkannt werden mit den dazugehörigen Fehlermeldungen.

- Verschiedene Stack-Längen an der gleichen Programmstelle: `Inconsistent stack height`
- Auslesen eines Wertes aus einem uninitialisierten Register: `Accessing value from uninitialized register`
- Stackhöhe überschreitet die Maximalhöhe: `Stack size too large`
- Stack ist leer während versucht wird, ein Element vom Stack zu holen: `Unable to pop operand off an empty stack`
- Ein Integer wird auf dem Stack erwartet, aber ein Wert eines anderen Typs wird vorgefunden: `Expecting to find integer on stack`
- Eine Objektreferenz wird auf dem Stack erwartet, aber ein Wert eines anderen Typs wird vorgefunden: `Expecting to find object/array on stack`
- Ein Wert soll einem Feld zugewiesen werden, obwohl das entsprechende Objekt kein solches Feld hat: `Incompatible type for getting or setting field`

  (Die gleiche Fehlermeldung tritt auf, wenn eine Methode aufgerufen wird, die nicht existiert.)
- Eine Methode gibt einen Rückgabewert vom falschen Typ zurück: `Wrong return type in function`
- Stack-Inhalte von inkompatiblen Typ an der gleichen Programmstelle: `Mismatched stack types`

Bemerkung: Ein Java Class-File `program.class` kann mit dem Befehl `javap -c program` bzw. `javap -verbose program` disassembliert werden, womit der erzeugte Bytecode betrachtet werden kann. Außderdem gibt es einen Java Bytecode Assembler namens `jasmin` (`http://jasmin.sourceforge.net/`), mit dem man eigenen (auch fehlerhaften) Bytecode erzeugen und in Java Classfiles umwandeln kann. Auf diese Weise kann man den Bytecode Verifier testen und Fehlermeldungen erhalten, die Laufzeitfehler anzeigen.

4.2 Der Java Bytecode Verifier

Analog zu Kapitel 3 betrachten wir ein JVM Programm S als eine Folge von Blöcken der Form $\ell:\ cmd$, wobei ℓ eine Zeilennummer zwischen 1 und k angibt und cmd ein Befehl aus obiger Liste ist. Es gilt $init(S) = 1$, $final(S) = \{k\}$ und die Flussrelation $flow(S)$ ist folgendermaßen definiert: für jeden Block $\ell:\ \mathsf{CmpEq}\ q$ zwei Paare $(\ell, \ell+1)$, (ℓ, q) und für jeden sonstigen Block $\ell:\ cmd$ mit $\ell < k$ und $cmd \neq \mathsf{Return}$ ein Paar $(\ell, \ell+1)$.

Wir machen eine Vorwärtsanalyse mit Supremumsbildung (Bildung der kleinsten oberen Schranke mit $\sqcup$). Wir definieren dazu einen Verband T als Property Space. Zunächst betrachten wir die Klassenhierarchie von Java. Da es bei Java keine Mehrfachvererbung gibt und eine feste Klasse Class existiert, von der alle anderen Klassen erben, ist die Klassenhierarchie ein Baum mit Wurzel Class. Für zwei Klassen C und D schreiben wir $C \sqsubseteq_c D$ genau dann, wenn C von D

abgeleitet wurde. Des weiteren legen wir folgende Ordnung $\sqsubseteq_t$ auf den zu Beginn von Abschnitt 4.1 definierten Typen Ty fest. Es gilt

$$\begin{aligned} ty_1 \sqsubseteq_t ty_2 \quad \Longleftrightarrow \quad & ty_1 = ty_2 \ \vee \\ & (ty_1 = \mathsf{NullT} \ \wedge \ \mathit{isref}(ty_2)) \ \vee \\ & (\mathit{isref}(ty_1) \ \wedge \ ty_2 = \mathsf{Class}) \ \vee \\ & (ty_1 = \mathsf{Class}\ C_1 \ \wedge \ ty_2 = \mathsf{Class}\ C_2 \ \wedge \ C_1 \sqsubseteq_c C_2) \end{aligned}$$

Alle anderen Typen stehen nicht in Relation, z.B. sind Integer und NullT unvergleichbar.

Warum die Bedingung $\mathsf{NullT} \sqsubseteq_t \mathsf{Class}\ A$ für jede Klasse A gelten soll, muss man noch kurz erklären. Je kleiner ein Element bezüglich $\sqsubseteq_t$ ist, desto genauer ist es festgelegt. D.h. insbesondere, dass auf einem kleineren Element mindestens die Operationen (Methodenaufrufe, Zuweisungen an Felder, etc.) möglich sein müssen, wie bei einem größeren Element. Tatsächlich ist es so, dass bei dem Null-Pointer jede beliebige Methode aufgerufen werden kann und jedem beliebigen Feld ein Wert zugewiesen werden kann. Dies soll vom Bytecode Verifier nicht als Fehler angesehen werden. Konkret wird dann an diesen Stellen eine Exception verursacht, die von einem Exception-Handler behandelt werden muss. Solche Exception-Handler kann man in realem Java-Bytecode beschreiben und sie können mit in die Dateflussanalyse integriert werden. In unserem kleinen Ausschnitt des Java-Bytecodes betrachten wir keine Exceptions, so dass wir im folgenden nicht weiter auf dieses Phänomen eingehen werden.

Wir machen folgende Annahmen über die Felder und Methoden von Objekten: falls eine Klasse C ein Feld *fname* vom Typ ty hat und $D \sqsubseteq_c C$ gilt, so muss auch die Klasse D ein Feld *fname* vom Typ ty haben. Ebenso muss D eine Methode *mname* mit Parameter ty_1 und Rückgabewert ty_2 haben, falls C eine solche Methode besitzt. Daraus kann man folgendes schließen: zunächst einmal kann man statt eines Objektes der Klasse C immer auch ein Objekt der Klasse D verwenden, ohne dass sich ein Laufzeitfehler ergibt. Außerdem kann man, da Informationen über die Existenz von Feldern bzw. Methoden bei bestimmten Klassen global vorhanden sind, sehr einfach überprüfen, ob die Befehle Invoke, Getfield und Putfield tatsächlich existierende Felder und Methoden ansprechen. Dies kann durch einen einmaligen Durchlauf des Programms entschieden werden und wir werden dies in der folgenden Analyse nicht mehr berücksichtigen.

Der Property Space T, mit dem wir die Analyse durchführen besteht aus folgender Menge T und einer Ordnung $\sqsubseteq_T$:

$$T = \{\mathit{None}, \mathit{Err}\} \cup (\mathit{Ty}^* \times (\{\mathit{Undef}\} \cup \mathit{Ty})^m)$$

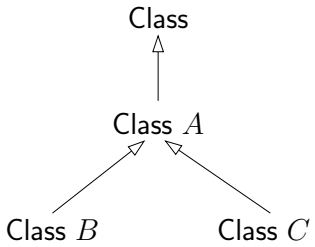
Dabei sind *None* und *Err* zusätzlich eingeführte Elemente, die $\bot$ und $\top$ im Verband darstellen. Dabei annotiert *None* einen noch nicht erreichten Befehl und *Err* einen Befehl, bei dem ein fehlerhafter Stack- oder Registerzustand auftritt. Die weiteren Elemente von T bezeichnen die Typen des Stacks, dargestellt durch eine beliebig lange Liste von Typen (Ty^*), und die Typen der Register, dargestellt durch ein m-Tupel von Typen, wobei auch *Undef* (Register darf in diesem Befehl nicht gelesen werden) auftauchen darf. Dabei ist m die (gegebene) Anzahl der Register. Die Menge Ty^* enthält alle Wörter beliebiger Menge über der Menge Ty, wobei wir im Folgenden Wörter als Listen in eckigen Klammern darstellen werden.

Wir definieren nun folgende partielle Ordnung $\sqsubseteq_T$:

$$\begin{aligned} \mathit{None} \sqsubseteq_T t \quad & \text{für alle } t \in T \\ t \sqsubseteq_T \mathit{Err} \quad & \text{für alle } t \in T \\ (st_1, reg_1) \sqsubseteq_T (st_2, reg_2) \quad \Longleftrightarrow \quad & st_i = [s^i_1, \ldots, s^i_{p_i}] \ \wedge \ reg_i = [r^i_1, \ldots, r^i_m] \quad \text{für } i = 1,2 \ \wedge \\ & p_1 = p_2 \ \wedge \\ & \forall\, 1 \le j \le p_1 : (s^1_j \sqsubseteq_t s^2_j) \ \wedge \\ & \forall\, 1 \le j \le m : (r^2_j = \mathit{Undef} \vee r^1_j \sqsubseteq_t r^2_j) \end{aligned}$$

Aufgabe 4.2.1 Welche der oben definierten Ordnungen $\sqsubseteq_c$, $\sqsubseteq_t$, $\sqsubseteq_T$ sind Verbände, d.h., in welchen Fällen ist die Eigenschaft erfüllt, dass jede Teilmenge L der der Ordnung zugrundeliegenden Menge eine kleinste obere Schranke und eine größte untere Schranke besitzt?

Aufgabe 4.2.2 Wir betrachten die unten dargestellte Klassenhierarchie und die daraus abgeleitete Ordnung $\sqsubseteq_c$. Außerdem sei die Anzahl der Register $m = 2$.



(a) Es sei $t = ([\mathsf{Integer}, \mathsf{Class}\ A], [\mathsf{NullT}, \mathsf{Class}\ C])$. Für welche der unten angegebenen $t' \in T$ gilt $t \sqsubseteq_T t'$?

i) $t' = ([\mathsf{Integer}], [\mathsf{NullT}, \mathsf{Class}\ C])$

ii) $t' = ([\mathsf{Integer}, \mathsf{Class}], [\mathsf{Class}\ A, \mathsf{Class}\ B])$

iii) $t' = ([\mathsf{Integer}, \mathsf{Class}], [\mathsf{Undef}, \mathsf{Class}\ A])$

iv) $t' = ([\mathsf{NullT}, \mathsf{Class}\ A], [\mathsf{NullT}, \mathsf{Class}\ C])$

v) $t = ([\mathsf{Integer}, \mathsf{Class}\ A], [\mathsf{Integer}, \mathsf{Class}\ C])$

(b) Bestimmen Sie für die oben angegebenen Elemente $t, t' \in T$ jeweils das Supremum $t \sqcup t'$.

Aufgabe 4.2.3 Argumentieren Sie, dass der Verband T die Ascending Chain Condition erfüllt, obwohl er unendlich viele Elemente enthält.

Als initialen Analysewert setzen wir

$$\iota = ([], \underbrace{[\mathsf{Class}\ C, p, \mathit{Undef}, \ldots, \mathit{Undef}]}_{\text{Länge } m}),$$

wobei C die Klasse ist, zu der das Objekt gehört, dessen Methode aufgerufen wird. Des weiteren ist p der Typ des Parameters der Funktion. Für alle übrigen Register wird *Undef* als Analysewert benutzt, um zu kennzeichnen, dass dieses Register hier nicht gelesen werden darf, da es nicht initialisiert ist (siehe auch die Typregel für Load in Tabelle 4.1).

Nun fehlt nur noch die Definition der Transferfunktionen f_ℓ. Diese sind etwas komplexer als die meisten der in Kapitel 3 vorkommenden Transferfunktionen und führen jeweils folgende Operationen durch: zunächst wird überprüft, ob der in jedem Block ankommende Analysewert $\mathsf{A}_\circ(\ell)$ konsistent mit dem aufzuführenden Befehl ist. Bei einer Pop-Operation muss beispielsweise überprüft werden, ob der Stack nicht-leer ist, bei einer Push-Operation, ob der Stack nicht bereits maximal aufgefüllt ist. Es muss ebenso überprüft werden, ob die aufzurufenden Methoden die korrekten Parameter- und Resultat-Typen haben, etc. Diese Überprüfung nehmen wir mit Hilfe von Typregeln vor. Falls die Überprüfung nicht gelingt, so gilt $\mathsf{A}_\bullet(\ell) = \mathit{Err}$. Anderenfalls muss der Ausgangswert berechnet werden, i.a. durch Manipulation der Stacktypen und der Stacklänge.

Zunächst einmal betrachten wir die verwendeten Typregeln (siehe Tabelle 4.1). Wir schreiben $(st, reg) \vdash cmd$, falls die Typen des Stacks und der Register mit dem Kommando *cmd* übereinstimmen. Für kein Kommando *cmd* gilt $\mathit{None} \vdash \mathit{cmd}$ bzw. $\mathit{Err} \vdash \mathit{cmd}$.

In obiger Tabelle bezeichnet $\circ$ die Konkatenation von Listen (siehe auch Kapitel 2 zur Notation).

Des weiteren verwenden wir Funktionen der Form $\mathsf{transfer}(\mathit{cmd}, (st, reg))$ für die Beschreibung der Transferfunktionen (siehe Tabelle 4.2).

$$\frac{|st| < max, reg!n \neq Undef}{(st, reg) \vdash \mathsf{Load}\ n} \qquad \frac{|st| > 0}{(st, reg) \vdash \mathsf{Store}\ n} \qquad \frac{|st| < max}{(st, reg) \vdash \mathsf{AConst\_Null}} \qquad \frac{|st| < max}{(st, reg) \vdash \mathsf{IConst}\ i}$$

$$\frac{st = [\mathsf{Integer}, \mathsf{Integer}] \circ st'}{(st, reg) \vdash \mathsf{IAdd}} \qquad \frac{st = [ty_0] \circ st',\ ty_0 \sqsubseteq_t \mathsf{Class}\ cname}{(st, reg) \vdash \mathsf{Getfield}\ fname\ ty\ cname}$$

$$\frac{st = [ty_v, ty_0] \circ st',\ ty_v \sqsubseteq_t ty,\ ty_0 \sqsubseteq_t \mathsf{Class}\ cname}{(st, reg) \vdash \mathsf{Putfield}\ fname\ ty\ cname}$$

$$\frac{|st| < max}{(st, reg) \vdash \mathsf{New}\ cname} \qquad \frac{st = [ty_a, ty_0] \circ st',\ ty_0 \sqsubseteq_t \mathsf{Class}\ cname,\ ty_a \sqsubseteq_t ty_1}{(st, reg) \vdash \mathsf{Invoke}\ cname\ mname\ ty_1\ ty_2}$$

$$\frac{st = [ty_1, ty_2] \circ st',\ ty_1 = ty_2 = \mathsf{Integer} \vee (\mathsf{isref}(ty_1) \wedge \mathsf{isref}(ty_2))}{(st, reg) \vdash \mathsf{CmpEq}\ q} \qquad \frac{st = [ty] \circ st',\ ty \sqsubseteq_t ty_r}{(st, reg) \vdash \mathsf{Return}}$$

Tabelle 4.1: Typregeln für die Transferfunktionen des Java Bytecode Verifiers

$$\begin{aligned}
\mathsf{transfer}(\mathsf{Load}\ n, (st, reg)) &= ([reg!n] \circ st, reg) \\
\mathsf{transfer}(\mathsf{Store}\ n, (st, reg)) &= (rest(st), reg[n \mapsto first(st)]) \\
\mathsf{transfer}(\mathsf{AConst_Null}, (st, reg)) &= ([\mathsf{NullT}] \circ st, reg) \\
\mathsf{transfer}(\mathsf{IConst}\ i, (st, reg)) &= ([\mathsf{Integer}] \circ st, reg) \\
\mathsf{transfer}(\mathsf{IAdd}, (st, reg)) &= ([\mathsf{Integer}] \circ rest^2(st), reg) \\
\mathsf{transfer}(\mathsf{Getfield}\ fname\ ty\ cname, (st, reg)) &= ([ty] \circ rest(st), reg) \\
\mathsf{transfer}(\mathsf{Putfield}\ fname\ ty\ cname, (st, reg)) &= (rest^2(st), reg) \\
\mathsf{transfer}(\mathsf{New}\ cname, (st, reg)) &= ([\mathsf{Class}\ cname] \circ st, reg) \\
\mathsf{transfer}(\mathsf{Invoke}\ cname\ mname\ ty_1\ ty_2, (st, reg)) &= ([ty_2] \circ rest^2(st), reg) \\
\mathsf{transfer}(\mathsf{CmpEq}\ q, (st, reg)) &= (rest^2(st), reg) \\
\mathsf{transfer}(\mathsf{Return}, (st, reg)) &= (st, reg)
\end{aligned}$$

Tabelle 4.2: Transferfunktionen des Java Bytecode Verifiers

Dabei bezeichnet $l!n$ das n-te Element der Liste l und $l[n \mapsto e]$ bezeichnet die Liste, die entsteht, wenn das n-te Element von l mit e überschrieben wird. Mit *first*(l) bezeichnen wir das erste Element einer Liste l und mit *rest*(l), die Liste, die entsteht, wenn das erste Element von l entfernt wird.

Die Transferfunktionen f_ℓ werden nun folgendermaßen definiert:

$$f_\ell(t) = \begin{cases} \mathsf{transfer}(cmd, t) & \text{falls } t = (st, reg),\ (st, reg) \vdash cmd \text{ und } block(\ell) = cmd \\ None & \text{falls } t = None \\ Err & \text{sonst.} \end{cases}$$

Das Ergebnis der Datenflussanalyse wird auch im Java Bytecode Verifier mit einer Variante des Worklist-Algorithmus aus Abschnitt 3.3 berechnet.

Theorem 4.2.4 (Korrektheit des Java Bytecode Verifiers) *Falls für das Analyseergebnis des Java Bytecode Verifiers gilt:* $\mathsf{A}_\circ(\ell) \neq \mathsf{Err}$ *und* $\mathsf{A}_\bullet(\ell) \neq \mathsf{Err}$ *für alle vorkommenden Labels* ℓ*, so erzeugt das analysierte Programm keine Laufzeitfehler.*

Dabei sind Laufzeitfehler die Verwendung von Werten mit falschem Typ, ein Rückgabewert von falschem Typ, verschiedene Stack-Längen am gleichen Programmpunkt, sowie Stack-Überlauf und der Versuch des Entfernens von Elementen vom leeren Stack (siehe auch die Liste am Ende von Abschnitt 4.1).

Es ist allerdings zu beachten, dass Analyse-Ergebnisse, die für ein bestimmtes Register den Wert *Undef* enthalten, durchaus möglich sind. Bereits der initiale Analysewert hat diese Eigenschaft. Dies bedeutet nur, dass das Register nicht gelesen werden darf, da es nicht initalisiert ist, es bezeichnet jedoch keinen eigentlichen Fehler.

Aufgabe 4.2.5 Spielen Sie für folgende Programme Java Bytecode Verifier und bestimmen Sie das Analyseergebnis. Verläuft die Analyse jeweils erfolgreich im Sinne von Theorem 4.2.4?

Verwenden Sie dazu die Klassenhierarchie aus Aufgabe 4.2.2 und den initalen Wert $\iota = ([], [\mathsf{Class}\ A, \mathsf{Class}\ B, \mathit{Undef}, \ldots, \mathit{Undef}])$. Nehmen Sie außerdem an, dass die Klassen zumindest folgende Felder und Methoden besitzen: alle drei Klassen besitzen eine Methode m mit einem Parameter vom Typ Integer und einem Rückgabewert vom Typ Class C. Außerdem besitzt die Klasse C ein Feld f vom Typ Integer.

Beachten Sie außerdem, dass die Nummerierung der Register bei 0 beginnt.

(a) Die Methode hat folgendes Aussehen und der Rückgabewert soll vom Typ Class C sein.

```
1:  Load 1
2:  IConst 1
3:  Invoke A m Integer C
4:  Store 0
5:  Load 0
6:  Getfield f Integer C
7:  IConst 0
8:  CmpEq 1
9:  Load 0
10: Return
```

(b) Die Methode hat folgendes Aussehen und der Rückgabewert soll vom Typ Integer sein.

```
1:  Load 0
2:  Getfield f Integer C
3:  Return
```

(c) Die Methode hat folgendes Aussehen und der Rückgabewert soll vom Typ Integer sein.

```
1: IConst 3
2: IConst 5
3: IAdd
4: Store 0
5: Load 0
6: IConst 8
7: CmpEq 1
8: Return
```

Bemerkung: Seit Java 7 (Bytecode-Version ≥ 50) gibt es die Neuerung, dass der Bytecode an den angesprungenen Stellen mit den entsprechenden Stack- und Registertypen annotiert werden muss (sogenannte Stack Map Frames). `jasmin` unterstützt solche Annotationen. Damit die Annotationen nicht zu komplex werden, kann ein Stack Map Frame die Typen des vorhergehenden Frames teilweise wiederverwenden.

Dies führt zu effizienterer Verifikation, da der Worklist-Algorithmus nicht mehrmals iteriert werden muss. Vielmehr muss der Code nur einmal linear durchlaufen werden, wobei überprüft wird, ob die Annotationen an den Sprungstellen korrekt ist.

Diese Änderung hat jedoch such zu Kritik geführt, da es dadurch sehr komplex geworden ist, Bytecode selbst zu schreiben bzw. zu generieren.

Der Stack Map Table, der wiederum die Stack Map Frames enthält, wird durch das Kommando `javap -verbose program` angezeigt.

Kapitel 5

Abstrakte Interpretation

5.1 Grundlagen der abstrakten Interpretation

Üblicherweise können Programme nicht vollständig ausgetestet werden, weil es unendliche viele Eingabewerte (z.B. alle ganzen Zahlen) gibt und auch die Variablen Belegungen aus einem unendlich großen Werteraum haben können. Der Ansatz der abstrakten Interpretation ist es, das Programm nicht auf den eigentlichen Werten, sondern auf Abstraktionen der Datentypen auszuführen. Beispielsweise könnte man die ganzen Zahlen durch *even* und *odd* abstrahieren, je nachdem ob die Zahl gerade oder ungerade ist. Eine andere Möglichkeit wäre es, die ganzen Zahlen in Äquivalenzklassen zu unterteilen, je nachdem ob sie kleiner, gleich oder größer Null sind $(-, \mathbf{0}, +)$.

Auf diesen Abstraktionen der Datentypen wird dann das Progamm ausgeführt. Falls es davon nicht viele gibt, so ist es prinzipiell möglich, das Programm auf allen Eingaben zu testen. Allerdings ist es wichtig, die Operationen auf den abstrakten Werten so zu definieren, dass tatsächlich alle Werte, die von dem jeweiligen abstrakten Wert repräsentiert werden, berücksichtigt werden. Nehmen wir beispielsweise an, dass ein Programm statt auf Integer-Werten (ganzen Zahlen) auf Werten der Form $I \subseteq \{-, \mathbf{0}, +\}$ ausgeführt wird. Außerdem sei die Operation $op : \mathbb{Z} \to \mathbb{Z}$ folgendermaßen definiert: $op(z) = z - 2$. Die Frage ist nun, wie beispielsweise der Wert $op^{\#}(\{+\})$ aussehen soll, wenn $op^{\#}$ die abstrakte Variante der Operation op ist. Der abstrakte Wert $+$ repräsentiert unter anderem die Werte 1, 2 und 3. Bei Subtraktion von 2 ergibt sich dann ein Wert der entweder kleiner, gleich oder größer als Null ist. Es muss also gelten: $op^{\#}(\{+\}) = \{-, \mathbf{0}, +\}$.

Ein weiteres einfaches Beispiel für abstrakte Interpretation ist auch ein Verfahren, mit dem man die Korrektheit von arithmetischen Berechnungen austesten kann. Beispielsweise wollen wir testen, ob die Berechnung

$$373 * 8847 + 12345 \stackrel{?}{=} 3312266$$

korrekt ist. Wir nutzen dazu die Tatsache aus, dass eine Zahl durch 9 teilbar ist, genau dann wenn ihre Quersumme durch 9 teilbar ist. Man kann also die Teilbarkeit durch 9 durch wiederholte Quersummenbildung überprüfen. Insbesondere gilt, dass eine Zahl den gleichen Divisionsrest durch 9 besitzt wie ihre Quersumme, das liegt an der Beziehung $[(10 \cdot a + b) \mod 9] = [(a + b) \mod 9]$. Des weiteren gilt $[a \cdot b \mod 9] = [((a \mod 9) \cdot (b \mod 9)) \mod 9]$ und $[a + b \mod 9] = [((a \mod 9) + (b \mod 9)) \mod 9]$.

Wir bilden also die iterierten Quersummen der vier obigen Zahlen und erhalten dabei 4 (für 373), 9 (für 8847), 6 (für 12345) und 5 (für 3312266). Wir führen nun die Berechnung auf den Quersummen durch und erhalten $4 \cdot 9 + 6 = 42$, Quersumme 6. Wir erhalten also auf beiden Seiten nicht dieselbe Quersumme, was aber der Fall sein müsste, wenn die Berechnung korrekt wäre. Wir wissen daher, dass obige Gleichung nicht gilt!

Geeignete Literatur zum Thema abstrakte Interpretation ist [Cou96, JN95, CC79, CC77].

5.1.1 Galois-Verbindungen

Wir definieren nun allgemein, welche Form die Abstraktionsabbildung α haben soll, die jeden Wert auf seinen zugehörigen abstrakten Wert abbildet. Dazu definieren wir neben α noch eine zweite Abbildung, die sogenannte Konkretisierung γ, die einem abstrakten Wert alle konkreten Werte zuordnet, für die er steht. Für den Fall der Abstraktionen *even* und *odd* würden die Abbildungen α_{eo} und γ_{eo} beispielsweise folgendermaßen aussehen:

Beispiel 5.1.1

$$
\begin{array}{rcl}
\alpha_{eo}\colon \mathcal{P}(\mathbb{Z}) & \to & \mathcal{P}(\{even, odd\}) \\
\alpha_{eo}(Z) & = & \begin{cases} \emptyset & \text{falls } Z = \emptyset \\ \{even\} & \text{falls } Z \subseteq \{\ldots,-4,-2,0,2,4,\ldots\},\ Z \neq \emptyset \\ \{odd\} & \text{falls } Z \subseteq \{\ldots,-3,-1,1,3,\ldots\},\ Z \neq \emptyset \\ \{even, odd\} & \text{sonst} \end{cases} \\
\gamma_{eo}\colon \mathcal{P}(\{even, odd\}) & \to & \mathcal{P}(\mathbb{Z}) \\
\gamma_{eo}(M) & = & \begin{cases} \emptyset & \text{falls } M = \emptyset \\ \{\ldots,-4,-2,0,2,4,\ldots\} & \text{falls } M = \{even\} \\ \{\ldots,-3,-1,1,3,\ldots\} & \text{falls } M = \{odd\} \\ \mathbb{Z} & \text{sonst} \end{cases}
\end{array}
$$

Abstraktion und Konkretisierung sollten folgendermaßen zusammenpassen: wird eine Menge Z von konkreten Werten erst abstrahiert und dann konkretisiert, so muss eine größere (oder gleiche) Menge als zuvor entstehen, das drückt aus, dass α über-, aber nicht unter-approximieren darf. Es muss also $Z \subseteq \gamma(\alpha(Z))$ gelten. In unserem Beispiel gilt $\gamma_{eo}(\alpha_{eo}(\{1,3,7\})) = \gamma_{eo}(\{odd\}) = \{,-3,-1,1,3,\ldots\} \supseteq \{1,3,7\}$. Umgekehrt sollte für jede Menge M von abstrakten Werten gelten: $\alpha(\gamma(M)) \subseteq M$, d.h. die Konkretisierung und anschließende Abstraktion ist nicht ungenauer als der ursprüngliche Wert.

Ein solches Paar $\langle \alpha, \gamma \rangle$ bezeichnet man auch als Galois-Verbindung. Neben Galois-Verbindungen zwischen Potenzmengen-Verbänden kann man Galois-Verbindungen auch allgemein auf Verbänden definieren.

Definition 5.1.2 (Galois-Verbindung) *Gegeben seien zwei Verbände $(L, \sqsubseteq)$ und $(M, \sqsubseteq)$. Ein Paar $\langle \alpha, \gamma \rangle$ von monotonen Funktionen $\alpha\colon L \to M$, $\gamma\colon M \to L$ heißt* Galois-Verbindung *genau dann, wenn gilt:*

$$\forall\, l \in L\colon l \sqsubseteq \gamma(\alpha(l)) \tag{5.1}$$

$$\forall\, m \in M\colon \alpha(\gamma(m)) \sqsubseteq m \tag{5.2}$$

Obige Definition wird in Abbildung 5.1 für Potenzmengenverbände $L = \mathcal{P}(A)$ und $M = \mathcal{P}(B)$ graphisch dargestellt.

Beispiel 5.1.3 (Intervall-Rechnung) Eine andere Galois-Verbindung bildet eine Menge von reellen Zahlen auf das kleinste sie umschließende Intervall ab. Wir verwenden die Menge $\mathbb{R}^\infty = \mathbb{R} \cup \{-\infty, \infty\}$, d.h. die reellen Zahlen angereichert mit $-\infty$ und ∞ und betrachten die folgenden beiden Verbände:

- $(\mathcal{P}(\mathbb{R}^\infty), \subseteq)$, d.h., die Potenzmenge mit der üblichen Inklusionsordnung.
- $(\mathbb{R}^\infty \times \mathbb{R}^\infty, \sqsubseteq)$, wobei $(r_1, r_2) \sqsubseteq (r_1', r_2')$ genau dann, wenn $r_1' \leq r_1$ und $r_2 \leq r_2'$

Wir definieren $\alpha\colon \mathcal{P}(\mathbb{R}^\infty) \to \mathbb{R}^\infty \times \mathbb{R}^\infty$ und $\gamma\colon \mathbb{R}^\infty \times \mathbb{R}^\infty \to \mathcal{P}(\mathbb{R}^\infty)$ wie folgt:

$$
\begin{array}{rcl}
\alpha(R) & = & (\bigsqcap R, \bigsqcup R) \\
\gamma(r_1, r_2) & = & \begin{cases} [r_1, r_2] & \text{falls } r_1 \leq r_2 \\ \emptyset & \text{sonst} \end{cases}
\end{array}
$$

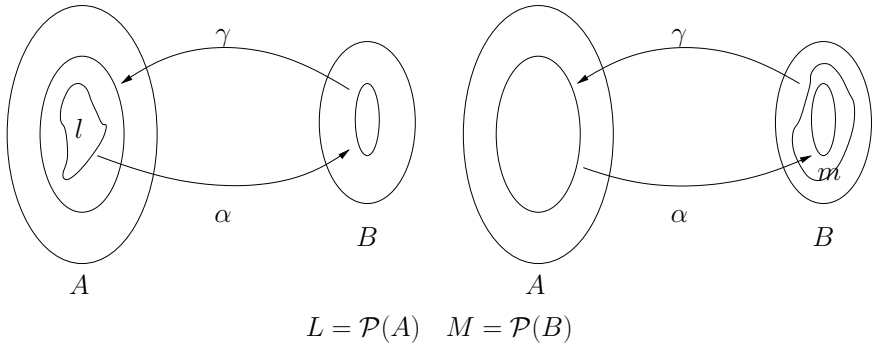


Abbildung 5.1: Graphische Darstellung der Bedingungen aus Definition 5.1.2.

wobei $[r_1, r_2]$ das Intervall zwischen r_1 und r_2 bezeichnet. Außerdem ist zu beachten, dass $\bigsqcap R$ und $\bigsqcup R$ das Infimum bzw. Supremum von R bezüglich der Relation $\leq$ auf reellen Zahlen bezeichnen. Falls R nicht nach unten beschränkt ist, so gilt $\bigsqcap R = -\infty$ und falls R nicht nach oben beschränkt ist, so gilt $\bigsqcup R = \infty$.

Beispiel 5.1.4 (Approximation durch konvexe Polyeder) Sei $\Sigma \subseteq (\mathbf{Var} \to \mathbb{R})$ eine Menge von Belegungen von Variablen mit reellen Zahlen. Sei außerdem $n = |\mathbf{Var}|$. Wir nehmen des weiteren an, dass $\mathbf{Var} = \{\mathtt{x_1}, \ldots, \mathtt{x_n}\}$. Dann kann man Σ als Menge von Punkten im $\mathbb{R}^n$ betrachten. Man approximiert sie durch das kleinste konvexe n-dimensionale Polyeder, das alle diese Punkte enthält. Abbildung 5.2 zeigt eine typische Situation für $n = 2$.

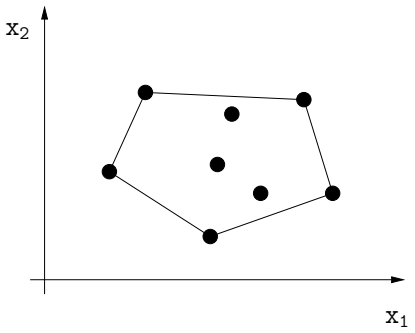


Abbildung 5.2: Abstraktion von Punkten durch Polyeder in der Ebene

Die Abstraktion α bildet damit ein Element aus $\mathcal{P}(\mathbf{Var} \to \mathbb{R})$ auf ein konvexes Polyeder $P \subseteq \mathbb{R}^n$ ab, das endlich darstellbar ist, beispielsweise indem man die Menge seiner Eckpunkte aufzählt. Dagegen bildet die Konkretisierung γ ein Polyeder P auf die Menge $\{\sigma\colon \mathbf{Var} \to \mathbb{R} \mid (\sigma(\mathtt{x_1}), \ldots, \sigma(\mathtt{x_n})) \in P\}$ ab.

Galois-Verbindungen haben bestimmte Eigenschaften, die wir im folgenden nutzen werden.

Satz 5.1.5 (Eigenschaften von Galois-Verbindungen (I)) *Sei $\langle \alpha, \gamma \rangle$ eine Galois-Verbindung mit $\alpha : L \to M$ und $\gamma : M \to L$, sowie $l \in L$ und $m \in M$. Dann gilt:*

$$\alpha(l) \sqsubseteq m \iff l \sqsubseteq \gamma(m).$$

Beweis: Es gelte $\alpha(l) \sqsubseteq m$ für zwei Verbandselemente $l \in L$, $m \in M$. Dann gilt mit der Monotonie von γ:

$$l \overset{(5.1)}{\sqsubseteq} \gamma(\alpha(l)) \sqsubseteq \gamma(m).$$

Nehmen wir nun an, dass umgekehrt $l \sqsubseteq \gamma(m)$ für $l \in L$, $m \in M$. Wir folgern nun mit der Monotonie von α:

$$\alpha(l) \sqsubseteq \alpha(\gamma(m)) \overset{(5.2)}{\sqsubseteq} m.$$

□

Satz 5.1.6 (Eigenschaften von Galois-Verbindungen (II)) *Sei $\langle \alpha, \gamma \rangle$ eine Galois-Verbindung mit $\alpha : L \to M$ und $\gamma : M \to L$. Dann gilt:*

(i) Die Konkretisierung γ ist eindeutig durch α bestimmt und es gilt $\gamma(m) = \bigsqcup\{l \mid \alpha(l) \sqsubseteq m\}$.

(ii) Die Abstraktion α ist eindeutig durch γ bestimmt und es gilt $\alpha(l) = \bigsqcap\{m \mid l \sqsubseteq \gamma(m)\}$.

(iii) Es gilt $\alpha(\bigsqcup L') = \bigsqcup\{\alpha(l) \mid l \in L'\}$ für $L' \subseteq L$. Man sagt auch α ist vollständig additiv.

(iv) Es gilt $\gamma(\bigsqcap M') = \bigsqcap\{\gamma(m) \mid m \in M'\}$ für $M' \subseteq M$. Man sagt auch γ ist vollständig multiplikativ.

(v) Außerdem gibt es zu jeder vollständig additiven Funktion $\alpha : L \to M$ eine Funktion $\gamma : M \to L$, so dass $\langle \alpha, \gamma \rangle$ eine Galois-Verbindung ist. Des weiteren gibt es zu jeder vollständig multiplikativen Funktion $\gamma : M \to L$ eine Funktion $\alpha : L \to M$, so dass $\langle \alpha, \gamma \rangle$ eine Galois-Verbindung ist.

Beweis:

(i) Wir zeigen zunächst, dass gilt: $\gamma(m) = \bigsqcup\{l \mid \alpha(l) \sqsubseteq m\}$ für jede Galois-Verbindung $\langle \alpha, \gamma \rangle$. Damit ist dann auch gezeigt, dass γ eindeutig bestimmt ist, falls α bekannt ist.

Falls $\alpha(l) \sqsubseteq m$ gilt, dann folgt mit Satz 5.1.5, dass $l \sqsubseteq \gamma(m)$. Daraus folgt $\gamma(m) \sqsupseteq \bigsqcup\{l \mid \alpha(l) \sqsubseteq m\}$. Wegen $\alpha(\gamma(m)) \sqsubseteq m$ gilt außerdem $\gamma(m) \in \{l \mid \alpha(l) \sqsubseteq m\}$ und daraus folgt $\gamma(m) \sqsubseteq \bigsqcup\{l \mid \alpha(l) \sqsubseteq m\}$. Zusammengefasst ergibt sich dann $\gamma(m) = \bigsqcup\{l \mid \alpha(l) \sqsubseteq m\}$.

(ii) Analog zu (i).

(iii) Für jedes $l \in L'$ gilt $l \sqsubseteq \bigsqcup L'$ und mit der Monotonie von α folgt daraus $\alpha(l) \sqsubseteq \alpha(\bigsqcup L')$ und es ergibt sich $\bigsqcup\{\alpha(l) \mid l \in L'\} \sqsubseteq \alpha(\bigsqcup L')$.

Um zu zeigen, dass $\alpha(\bigsqcup L') \sqsubseteq \bigsqcup\{\alpha(l) \mid l \in L'\}$ gilt, reicht es nach Satz 5.1.5 zu zeigen, dass $\bigsqcup L' \sqsubseteq \gamma(\bigsqcup\{\alpha(l) \mid l \in L'\})$. Sei l' ein festes Element aus L', dann folgt aus $\alpha(l') \in \{\alpha(l) \mid l \in L'\}$, dass $\alpha(l') \sqsubseteq \bigsqcup\{\alpha(l) \mid l \in L'\}$. Es gilt $l' \overset{(5.2)}{\sqsubseteq} \gamma(\alpha(l')) \sqsubseteq \gamma(\bigsqcup\{\alpha(l) \mid l \in L'\})$. Da dies für jedes $l' \in L'$ gilt, folgt auch $\bigsqcup L' \sqsubseteq \gamma(\bigsqcup\{\alpha(l) \mid l \in L'\})$.

(iv) Analog zu (iii).

(v) Sei nun $\alpha\colon L \to M$ eine beliebige vollständig additive Funktion. Wir definieren $\gamma\colon M \to L$ mit $\gamma(m) = \bigsqcup\{l \mid \alpha(l) \sqsubseteq m\}$. Es ist nun zu zeigen, dass $\langle \alpha, \gamma \rangle$ eine Galois-Verbindung ist.

Zunächst ist zu zeigen, dass γ monoton ist. Dazu müssen wir beweisen, dass aus $m \sqsubseteq m'$ folgt: $\bigsqcup\{l \mid \alpha(l) \sqsubseteq m\} \sqsubseteq \bigsqcup\{l \mid \alpha(l) \sqsubseteq m'\}$. Sei $l \in L$ mit $\alpha(l) \sqsubseteq m$, dann gilt auch $\alpha(l) \sqsubseteq m'$, und daraus folgt $\{l \mid \alpha(l) \sqsubseteq m\} \subseteq \{l \mid \alpha(l) \sqsubseteq m'\}$ und damit auch $\bigsqcup\{l \mid \alpha(l) \sqsubseteq m\} \sqsubseteq \bigsqcup\{l \mid \alpha(l) \sqsubseteq m'\}$.

Außerdem gilt $\gamma(\alpha(l)) = \bigsqcup\{l' \mid \alpha(l') \sqsubseteq \alpha(l)\} \sqsupseteq l$ und außerdem $\alpha(\gamma(m)) = \alpha(\bigsqcup\{l \mid \alpha(l) \sqsubseteq m\}) = \bigsqcup\{\alpha(l) \mid \alpha(l) \sqsubseteq m\} \sqsubseteq m$, aufgrund der vollständigen Additivität von α.

Der Beweis für $\gamma\colon M \to L$ verläuft analog.

□

Oft kann man Galois-Verbindungen einfach mit Hilfe sogenannter Extraktionsfunktionen bestimmen. Sei $\beta : A_1 \to A_2$ eine Abbildung der Menge A_1 nach A_2. Wir können dann $\alpha : \mathcal{P}(A_1) \to \mathcal{P}(A_2)$ in Abhängigkeit von β wie folgt bestimmen:

$$\alpha(A_1') = \{\beta(a_1) \mid a_1 \in A_1'\}$$

für $A_1' \subseteq A_1$.

Aufgabe 5.1.7

(a) Gegeben sei eine Extraktionsfunktion β. Zeigen Sie, dass α - wie oben definiert - vollständig additiv ist und bestimmen Sie ein γ, so dass $\langle \alpha, \gamma \rangle$ eine Galois-Verbindung ist.

(b) Bestimmen Sie eine Extraktionsfunktion β, so dass die Galois-Verbindung $\langle \alpha_{eo}, \gamma_{eo} \rangle$ aus Beispiel 5.1.1 durch β erzeugt wird.

Mit Hilfe von Extraktionsfunktionen lassen sich leicht neue Galois-Verbindungen erzeugen. Häufig verwendete Extraktionsfunktionen sind beispielsweise:

Vorzeichen-Rechnung: $\beta : \mathbb{Z} \to \{-, \mathbf{0}, +\}$ mit

$$\beta(z) = \begin{cases} - & \text{falls } z < 0 \\ \mathbf{0} & \text{falls } z = 0 \\ + & \text{falls } z > 0 \end{cases}$$

Modulo-Rechnung: $\beta : \mathbb{Z} \to \{0, \ldots, n-1\}$ mit $\beta(z) = z \mod n$ für ein festes $n \in \mathbb{N}_0 \backslash \{0\}$.

Im folgenden wird L oft die Variablenbelegungen des zu analysierenden Programms repräsentieren. Hat das Programm beispielsweise drei Integer-Variablen x, y, z, dann setzen wir $L = \mathcal{P}(\mathbf{State})$ mit $\mathbf{State} = \{\sigma \mid \sigma : \{\mathtt{x}, \mathtt{y}, \mathtt{z}\} \to \mathbb{Z}\}$ (Menge aller Abbildungen der Variablen auf ganze Zahlen). Dagegen steht M für den Verband der abstrakten Werte, den wir manchmal auch mit **Abs** bezeichnen werden.

Wenn bereits eine Extraktionsfunktion $\beta \colon \mathbb{Z} \to A$ bekannt ist, dann ist es leicht, daraus eine Abstraktion zu erzeugen, die auf allen Elementen von $\mathcal{P}(\mathbf{State})$ arbeitet. Wir nennen dieses Vorgehen Liften der Extraktionsfunktion.

Definition 5.1.8 (Liften einer Extraktionsfunktion) *Sei* $\mathbf{State} = \mathbf{Var} \to \mathbb{Z}$ *die Menge aller möglichen Variablenbelegungen* σ *und sei* $\beta : \mathbb{Z} \to A$ *eine Extraktionsfunktion. Durch Liften von* β *in Bezug auf* **State** *entsteht die Abstraktion* $\alpha \colon \mathcal{P}(\mathbf{State}) \to \mathcal{P}(\mathbf{Var} \to A)$ *wie folgt:*

$$\alpha(\Sigma) = \{\beta \circ \sigma \mid \sigma \in \Sigma\}$$

für $\Sigma \subseteq \mathbf{State}$.

Man kann übrigens leicht zeigen, dass das oben definierte α vollständig additiv und damit Teil einer Galois-Verbindung ist.

5.1.2 Abstrakte Semantik

Wir müssen nun noch festlegen, wie sich Befehle und Operationen der Programmiersprache auf abstrakte Werte auswirken sollen.

Definition 5.1.9 (Sichere Approximation von Funktionen) *Sei $\langle \alpha, \gamma \rangle$ mit $\alpha \colon L \to M$ und $\gamma \colon M \to L$ eine Galois-Verbindung. Seien des weiteren $f \colon L^n \to L$ und $f^\# : M^n \to M$ n-stellige Funktionen. Die Funktion $f^\#$ heißt* sichere Approximation *von f genau dann wenn für alle $m_1, \ldots, m_n \in M$ gilt:*

$$\alpha(f(\gamma(m_1), \ldots, \gamma(m_n))) \sqsubseteq f^\#(m_1, \ldots, m_n).$$

Die Funktion $f^\#$ heißt genaueste sichere Approximation *von f genau dann, wenn in obiger Ungleichung die Gleichheit $=$ statt $\sqsubseteq$ verwendet wird.*

Das heißt, eine Funktion angewandt auf die abstrakten Werten ergibt ein ungenaueres Ergebnis, das auf jeden Fall das eigentliche Funktionsergebnis mit repräsentiert. Es wird fast immer sinnvoll, sein, zu verlangen, dass f und $f^\#$ monoton sind. Die Anwendung einer Funktion auf einen ungenaueren Wert sollte auch ein ungenaueres Ergebnis liefern.

Aufgabe 5.1.10 Zeigen Sie, dass gilt:

$$\begin{array}{ll} & \forall m_1, \ldots, m_n \in M \colon \alpha(f(\gamma(m_1), \ldots, \gamma(m_n))) \sqsubseteq f^\#(m_1, \ldots, m_n) \\ \iff & \forall l_1, \ldots, l_n \in L \colon \alpha(f(l_1, \ldots, l_n)) \sqsubseteq f^\#(\alpha(l_1), \ldots, \alpha(l_n)), \end{array}$$

falls f und $f^\#$ monoton sind.

Beispiel 5.1.11 Wir betrachten die in Abschnitt 5.1.1 vorgestellten Galois-Verbindungen und bestimmen dazu die (genaueste) sichere Approximation einiger Operationen. Die abstrakten Operatoren werden durch einen Kreis um die jeweilige Operation gekennzeichnet (z.B. $\oplus$ für Plus, $\ominus$ für Minus).

Even-Odd-Abstraktion: Hier gilt beispielsweise $\{even\} \ominus 1 = \{odd\}$, $\{odd\} \oplus 1 = \{even\}$, $\{even\} \odot \{odd\} = \{even\}$, etc.

Vorzeichen-Rechnung: In diesem Fall kommt Nichtdeterminismus zu tragen (siehe auch die Beispiele am Anfang dieses Kapitels), da wir den abstrakten Wert des Ergebnisses nicht mit Sicherheit bestimmen können. Es gilt beispielsweise $\{+\} \ominus 2 = \{-, \mathbf{0}, +\}$, $\{+\} \ominus 1 = \{\mathbf{0}, +\}$, $\{+\} \odot \{-\} = \{-\}$, etc.

Diese Rechnung und vor allem die Rolle der Konstanten muss eigentlich noch genauer erklärt werden: wir betrachten die Subtraktion der Konstante 1 als einstellige Funktion f, die, da wir ja hier mit Verbänden arbeiten, auf einer Menge von Zahlen operieren muss.

$$\begin{array}{rcl} f \colon \mathcal{P}(\mathbb{Z}) & \to & \mathcal{P}(\mathbb{Z}) \\ f(Z) & = & \{z - 1 \mid z \in Z\} \quad \text{wobei } Z \subseteq \mathbb{Z} \end{array}$$

Dazu definieren wir eine Abstraktion $f^\#$ dieser Funktion mit:

$$\begin{array}{rcl} f^\# \colon \mathcal{P}(\{-, 0, +\}) & \to & \mathcal{P}(\{-, 0, +\}) \\ f^\#(A) & = & \{- \mid \text{falls } 0 \in A \vee - \in A\} \cup \{0, + \mid \text{falls } + \in A\} \quad \text{wobei } A \subseteq \{-, 0, +\} \end{array}$$

Man muss jetzt zeigen, dass für jedes A gilt: $\alpha(f(\gamma(A))) \subseteq f^\#(A)$. Dies ist erfüllt, beispielsweise ergibt sich für $A = \{+\}$:

$$\alpha(f(\gamma(A))) = \alpha(f(\{1, 2, ...\})) = \alpha(0, 1, 2, ...) = \{0, +\}$$

und außerdem $f^\#(A) = \{0, +\}$.

Wir betrachten nun noch die Funktion

$$\begin{array}{rcl} g \colon \mathcal{P}(\mathbb{Z}) & \to & \mathcal{P}(\mathbb{Z}) \\ g(Z) & = & \{z - 2 \mid z \in Z\} \quad \text{wobei } Z \subseteq \mathbb{Z} \end{array}$$

Die Funktion $f^\#$ ist jetzt jedoch keine Abstraktion von g mehr, denn es gilt für $A = \{+\}$:

$$\alpha(g(\gamma(A))) = \alpha(g(\{1, 2, ...\})) = \alpha(\{-1, 0, 1, 2, ...\}) = \{-, 0, +\},$$

aber $f^\#(A) = \{0, +\}$. Eine gültige Abstraktion ist aber die Funktion $g^\#$ mit

$$\begin{array}{rcl} g^\#: \mathcal{P}(\{-, 0, +\}) & \to & \mathcal{P}(\{-, 0, +\}) \\ g^\#(A) & = & \{- \mid \text{falls } 0 \in A \vee - \in A \vee + \in A\} \cup \{0, + \mid \text{falls } + \in A\}. \end{array}$$

Modulo-Rechnung: Bei der Addition erhalten wir im wesentlichen die Operationen der Gruppe $\mathbb{Z}_n$ ($\mathbb{Z}$ modulo n): $\{a\} \oplus \{b\} = \{(a + b) \mod n\}$ für $a, b \in \{0, \ldots, n - 1\}$. Und bei der Multiplikation erhält man analog $\{a\} \odot \{b\} = \{(a \cdot b) \mod n\}$.

Wir betrachten nun wieder die WHILE-Sprache aus Anhang A und beschreiben deren operationelle Semantik auf abstrakten Werten. Die Übergangsrelation kann nun allerdings nichtdeterministisch sein, denn ein Vergleich der Form `x = 0`, wobei `x` mit dem abstrakten Wert *even* belegt ist, kann entweder *true* oder *false* liefern.

Wir nehmen im folgenden an, dass der Verband $L = \mathcal{P}(\mathbf{State})$ die möglichen Variablenbelegungen des Programms repräsentiert, wobei **State** wie in Anhang A alle möglichen Funktionen der Form $\mathbf{Var} \to \mathbb{Z}$ enthält. Den Verband der abstrakten Werte bezeichnen wir mit **Abs**. Er kann durch Liften einer Extraktionsfunktion $\beta: \mathbb{Z} \to A$ entstanden sein und in diesem Fall gilt $\mathbf{Abs} = \mathcal{P}(\mathbf{Var} \to A)$, was aber nicht unbedingt Voraussetzung für die folgenden Definition ist.

Um die abstrakte Semantik eines Programms analog zu Definition 5.1.9 beschreiben zu können, benötigen wir zunächst folgende Ein-Schritt-Übergangsfunktion *next* auf Programmen.

Definition 5.1.12 (Übergangsfunktion *next*) *Es seien S, S' zwei* WHILE*-Programme und $\Sigma \subseteq \mathbf{State}$. Wir setzen*

$$next_{S,S'}: \mathcal{P}(\mathbf{State}) \to \mathcal{P}(\mathbf{State}), \qquad next_{S,\downarrow}: \mathcal{P}(\mathbf{State}) \to \mathcal{P}(\mathbf{State})$$

mit

$$next_{S,S'}(\Sigma) = \{\sigma' \mid \sigma \in \Sigma \ \wedge \ \langle S, \sigma\rangle \to \langle S', \sigma'\rangle\} \quad \textit{und} \quad next_{S,\downarrow}(\Sigma) = \{\sigma' \mid \sigma \in \Sigma \ \wedge \ \langle S, \sigma\rangle \to \sigma'\}.$$

Wir können nun die abstrakte Semantik von WHILE-Programmen definieren. Wir legen dazu fest, dass die abstrakte Semantik eine sichere Approximation der eigentlichen Semantik sein muss.

Definition 5.1.13 (Abstrakte Semantik) *Die abstrakte Semantik wird beschrieben durch eine Familie $next^\#_{S,S'}: \mathbf{Abs} \to \mathbf{Abs}$ von Funktionen für die gilt:*

$$\alpha(next_{S,S'}(\gamma(abs))) \sqsubseteq next^\#_{S,S'}(abs)$$

für $abs \in \mathbf{Abs}$ und alle WHILE*-Programme S, S' und $S' = \downarrow$.*

Wir schreiben auch $\langle S, abs\rangle \Longrightarrow \langle S', abs'\rangle$, falls $next^\#_{S,S'}(abs) = abs'$, und $\langle S, abs\rangle \Longrightarrow abs'$, falls $next^\#_{S,\downarrow}(abs) = abs'$.

Steht in obiger Definition ein Gleichheitszeichen anstelle von $\sqsubseteq$, so handelt es sich um die genaueste abstrakte Semantik. Unpräzisere Semantiken sind jedoch auch zulässig.

Wir betrachten diese genaueste Semantik an einem konkreten Beispiel und verwenden die Galois-Verbindung $\langle\alpha, \gamma\rangle$, die durch Liften der Extraktionsfunktion $\beta: \mathbb{Z} \to \{even, odd\}$ entsteht, wobei β analog zu Beispiel 5.1.1 definiert ist. Es gilt $\mathbf{Var} = \{\mathtt{n}\}$, d.h., die Variablenmenge enthält nur ein Element.

Wir betrachten nun die genaueste abstrakte Semantik anhand einiger Beispiele. Mit $[\mathtt{n} \mapsto a]$ bezeichnen wir dabei die Funktion, die die Variable n auf den Wert $a \in \{even, odd\}$ abbildet.

$$\begin{array}{rcl}
\langle [\texttt{n := 3*n+1}]^\ell, \{[\texttt{n} \mapsto \mathit{odd}]\}\rangle & \Longrightarrow & \{[\texttt{n} \mapsto \mathit{even}]\} \\
\langle [\texttt{n := 3*n+1}]^\ell, \{[\texttt{n} \mapsto \mathit{even}]\}\rangle & \Longrightarrow & \{[\texttt{n} \mapsto \mathit{odd}]\} \\
\langle [\texttt{n := 3*n+1}]^\ell, \{[\texttt{n} \mapsto \mathit{even}], [\texttt{n} \mapsto \mathit{odd}]\}\rangle & \Longrightarrow & \{[\texttt{n} \mapsto \mathit{even}], [\texttt{n} \mapsto \mathit{odd}]\} \\
\langle \texttt{while } [\texttt{n}\neq\texttt{1}]^\ell \texttt{ do } S \texttt{ od}, \{[\texttt{n} \mapsto \mathit{odd}]\}\rangle & \Longrightarrow & \{[\texttt{n} \mapsto \mathit{odd}]\} \\
\langle \texttt{while } [\texttt{n}\neq\texttt{1}]^\ell \texttt{ do } S \texttt{ od}, \{[\texttt{n} \mapsto \mathit{odd}]\}\rangle & \Longrightarrow & \langle S\texttt{;while } [\texttt{n} \neq \texttt{1}]^\ell \texttt{ do } S \texttt{ od}, \{[\texttt{n} \mapsto \mathit{odd}]\}\rangle \\
\langle \texttt{while } [\texttt{n}\neq\texttt{1}]^\ell \texttt{ do } S \texttt{ od}, \{[\texttt{n} \mapsto \mathit{even}]\}\rangle & \not\Longrightarrow & \{[\texttt{n} \mapsto \mathit{even}]\} \\
\langle \texttt{while } [\texttt{n}\neq\texttt{1}]^\ell \texttt{ do } S \texttt{ od}, \{[\texttt{n} \mapsto \mathit{even}]\}\rangle & \Longrightarrow & \langle S\texttt{;while } [\texttt{n}\neq\texttt{1}]^\ell \texttt{ do } S \texttt{ od}, \{[\texttt{n} \mapsto \mathit{even}]\}\rangle
\end{array}$$

5.1.3 Beispiel: Hailstone-Folge

Folgende Programmanalyse ist aus [JN95] entnommen.

Mit Hilfe obiger Semantik können wir nun folgendes kleine Beispielprogramm interpretieren, das die sogenannte Hailstone-Folge berechnet und terminiert, sobald der Wert `n` gleich 1 ist. Diese Folge ist auch noch unter vielen anderen Namen bekannt, z.B. als Collatz-Folge, $3n+1$-Problem, etc. (siehe auch `http://mathworld.wolfram.com/CollatzProblem.html`). Es ist übrigens nicht bekannt, ob dieses Programm für alle Eingabewerte terminiert. Für `n` = 11 lautet die Folge $11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, \ldots$ der Wert 1 wird also angenommen.

Programm 5.1.14 (HAILSTONE)

```
[skip]1;
while [n≠1]2 do
   if [even(n)]3
      then [n:=n/2]4;[skip]5
      else [n:=3*n+1]6;[skip]7
   fi
od
```

Die zusätzlichen Skip-Anweisungen dienen nur dazu, damit wir nachvollziehen können, wie die Variablenbelegungen zu Beginn und am Ende des Programms und nach den Zuweisungen aussehen.

Wenn wir annehmen, dass wir mit einer ungeraden Zahl n starten, d.h., falls der initiale abstrakte Wert gleich $[\texttt{n} \mapsto \mathit{odd}]$ ist, so erhalten wir die in Abbildung 5.3 dargestellten Übergänge. Wir zeichnen nur die Übergänge, die vom initialen abstrakten Wert aus erreichbar sind, und bei denen *abs* nicht die leere Menge ist.

Wie man dabei sehen kann, gehen hier, im Gegensatz zur Datenflussanalyse, die Analyseergebnisse auch in die Verzweigungen bei `while` und `if`-Anweisungen mit ein. Falls wir beispielsweise mit einem abstrakten Wert *even* für `n` die Abfrage der `while`-Schleife erreichen, so können wir sicher sein, dass wir die Schleife dort *nicht* verlassen. Ebenso sind, solange `n` nicht den abstrakten Wert $\{\mathit{even}, \mathit{odd}\}$ hat, die auf die `if`-Anweisung folgenden Kommandos festgelegt, da dort abgefragt wird, ob `n` gerade oder ungerade ist.

Man kann nun die Ergebnisse der obigen Analyse zusammenfassen, indem wir die möglichen abstrakten Werte von `n` am Anfang der jeweiligen Blöcken bestimmen. Dazu bilden wir jeweils das Supremum der in Abbildung 5.3 vorkommenden Analyseergebnisse.

1	2	3	4	5	6	7	Terminierung
$\{\mathit{odd}\}$	$\{\mathit{even}, \mathit{odd}\}$	$\{\mathit{even}, \mathit{odd}\}$	$\{\mathit{even}\}$	$\{\mathit{even}, \mathit{odd}\}$	$\{\mathit{odd}\}$	$\{\mathit{even}\}$	$\{\mathit{odd}\}$

Beispiel 5.1.15 Interpretieren Sie folgendes Programm auf den abstrakten Werten *even*, *odd*.

```
[skip]1;
while [n>0]2 do
   if [odd(n)]3
      then [n:=2*n]4;[skip]5
```

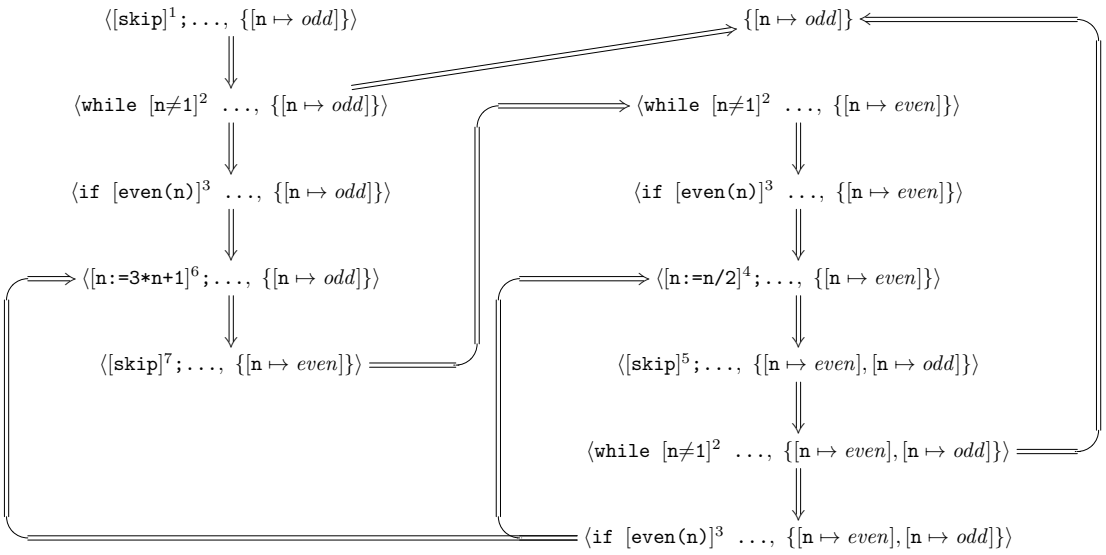


Abbildung 5.3: Übergänge des abstrakten HAILSTONE-Programms

```
      else [n:=n-2]⁶;[skip]⁷
   fi
od
```

5.1.4 Herleitung einer abstrakten Semantik

Es stellt sich nun das Problem, dass die in Anschluss an Definition 5.1.13 beschriebene genaueste abstrakte Semantik im allgemeinen gar nicht bestimmbar ist. Um dies einzusehen betrachten wir folgendes Beispiel: die Galois-Verbindung entspricht dem Liften der Vorzeichen-Rechnung, die im Anschluss an Aufgabe 5.1.7 vorgestellt wurde, auf $\mathbf{State} = (\{\mathtt{x}, \mathtt{y}, \mathtt{z}, \mathtt{n}\} \to \mathbb{Z})$.

Wir betrachten das Paar

$$\langle \texttt{if } [\mathtt{n}{>}2 \wedge \mathtt{x}^{\mathtt{n}} + \mathtt{y}^{\mathtt{n}} = \mathtt{z}^{\mathtt{n}}\ ]^{\ell} \texttt{ then } [\mathtt{n}{:=}1]^{\ell_1} \texttt{ else } [\mathtt{n}{:=}{-}1]^{\ell_2} \texttt{ fi}, \{[\mathtt{n}, \mathtt{x}, \mathtt{y}, \mathtt{z} \mapsto +]\}\rangle.$$

Zu entscheiden, ob bei der abstrakten Semantik n auf + oder − gesetzt wird, müsste man wissen, ob es eine Belegung der Variablen mit natürlichen Zahlen geben kann, so dass die Bedingung wahr ist. Nachdem der letzte Satz von Fermat inzwischen bewiesen wurde, wissen wir, dass es solche Zahlen nicht geben kann, wir können aber kaum erwarten, dass die abstrakte Semantik dies bei dieser und ähnlichen Bedingungen auch entscheiden kann.

Man kann sogar zeigen, dass es im allgemeinen unentscheidbar ist, ob ein (nicht-lineares) Gleichungssystem über den ganzen Zahlen (man sagt dazu auch diophantische Gleichungen) eine Lösung besitzt.

Wie wir also sehen können, ist die Berechnung der genauesten abstrakte Semantik im allgemeinen unmöglich, man kann jedoch relativ einfach eine ungenauere abstrakte Semantik herleiten. Wir beschränken uns auf den speziellen Fall, in dem die Galois-Abstraktion durch Liften einer Extraktionsfunktion β entstanden ist, es gilt also $\beta\colon \mathbb{Z} \to A$ und $\alpha\colon \mathcal{P}(\mathbf{State}) \to \mathcal{P}(\mathbf{Var} \to A)$ (siehe Definition 5.1.8). Wir setzen $\mathbf{Abs} = \mathcal{P}(\mathbf{Var} \to A)$.

Wir erweitern zunächst die Auswertungsfunktionen $\mathcal{A}$ und $\mathcal{B}$ aus Anhang A auf abstrakte Werte. Wir definieren im folgenden die Funktionen $\mathcal{A}_{abstr}\colon \mathbf{AExp} \times (\mathbf{Var} \to A) \to \mathcal{P}(A)$ und $\mathcal{B}_{abstr}\colon \mathbf{BExp} \times (\mathbf{Var} \to A) \to \{0, 1, 1/2\}$, wobei 0, 1, 1/2 Wahrheitswerte einer dreiwertigen Logik sind (siehe weiter unten).

Wir nehmen an, dass es zu jeder (n-stelligen) arithmetischen Operation $op\colon \mathbb{Z}^n \to \mathbb{Z}$ eine sichere Approximation $op^\#\colon \mathcal{P}(A)^n \to \mathcal{P}(A)$ gibt, mit folgender Eigenschaft: seien $A_1, \ldots, A_n \subseteq A$, dann muss $\beta(op(z_1, \ldots, z_n)) \in op^\#(A_1, \ldots, A_n)$ für alle $z_1 \in \beta^{-1}(A_1), \ldots, z_n \in \beta^{-1}(A_n)$ gelten.

Eine mögliche Definition für $op^\#$ ist also:

$$op^\#(A_1, \ldots, A_n) = \{\beta(op(z_1, \ldots, z_n)) \mid z_i \in \beta^{-1}(A_i), i \in \{1, \ldots, n\}\}.$$

Dies ist ähnlich zu Definition 5.1.9. Allerdings ist diese Definition insofern verschieden, als wir bei Definition 5.1.9 direkt auf den Verbandselementen operieren, während bei obiger Festlegung zumindest $op\colon \mathbb{Z}^n \to \mathbb{Z}$ nicht auf einem Verband operiert.

Um Boolesche Ausdrücke auswerten zu können, braucht man noch den Begriff der sicheren Approximation eines Prädikats. Wir werten im folgenden Prädikate auf einer dreiwertigen Logik aus, mit den Wahrheitswerten 0 (falsch), 1 (wahr) und 1/2 (unbestimmt). Diese dreiwertige Logik hat die in Abbildung 5.4 angegebenen Wahrheitstafeln.

$\wedge$	0	1	1/2
0	0	0	0
1	0	1	1/2
1/2	0	1/2	1/2

$\vee$	0	1	1/2
0	0	1	1/2
1	1	1	1
1/2	1/2	1	1/2

$\neg$	
0	1
1	0
1/2	1/2

Abbildung 5.4: Wertetabellen der dreiwertigen Logik

Definition 5.1.16 (Sichere Approximation von Prädikaten) *Sei $P\colon \mathbb{Z}^n \to \{true, false\}$ ein n-stelliges Prädikat und $P^\#\colon \mathcal{P}(A)^n \to \{0, 1, 1/2\}$. Die Funktion $P^\#$ heißt sichere Approximation von P genau dann wenn für alle $A_1, \ldots, A_n \subseteq A$ und alle $z_1 \in \beta^{-1}(A_1), \ldots, z_n \in \beta^{-1}(A_n)$ gilt:*

$$P(z_1, \ldots, z_n) \in val(P^\#(A_1, \ldots, A_n))$$

wobei

$$val(b) = \begin{cases} \{false\} & falls\ b = 0 \\ \{true\} & falls\ b = 1 \\ \{true, false\} & falls\ b = 1/2 \end{cases}.$$

Eine Möglichkeit, eine solche sichere Approximation eines Prädikats zu definieren, ist die folgende: für $A_1, \ldots, A_n \neq \emptyset$ definieren wir

$$P^\#(A_1, \ldots, A_n) = \begin{cases} 0 & \text{für alle } z_1 \in \beta^{-1}(A_1), \ldots, z_n \in \beta^{-1}(A_n) \text{ gilt } P(z_1, \ldots, z_n) = false \\ 1 & \text{für alle } z_1 \in \beta^{-1}(A_1), \ldots, z_n \in \beta^{-1}(A_n) \text{ gilt } P(z_1, \ldots, z_n) = true \\ 1/2 & \text{sonst} \end{cases}$$

Falls eine der Mengen A_i die leere Menge ist, kann man den Wert von $P^\#$ beliebig wählen.

Sei nun $\rho \in (\mathbf{Var} \to A)$. Die Funktionen $\mathcal{A}_{abstr}$ und $\mathcal{B}_{abstr}$ sind induktiv wie folgt definiert. Dabei ist $\mathtt{x}$ eine Variable, $z \in \mathbb{Z}$ eine ganze Zahl, b, b_1, b_2 boolesche Ausdrücke und die a_i arithmetische Ausdrücke.

$$\begin{aligned}
\mathcal{A}_{abstr}(\mathtt{x}, \rho) &= \{\rho(\mathtt{x})\} \\
\mathcal{A}_{abstr}(z, \rho) &= \{\beta(z)\} \\
\mathcal{A}_{abstr}(op(a_1, \ldots, a_n), \rho) &= op^\#(\mathcal{A}_{abstr}(a_1, \rho), \ldots, \mathcal{A}_{abstr}(a_n, \rho)) \\
\\
\mathcal{B}_{abstr}(P(a_1, \ldots, a_n), \rho) &= P^\#(\mathcal{A}_{abstr}(a_1, \rho), \ldots, \mathcal{A}_{abstr}(a_1, \rho)) \\
\mathcal{B}_{abstr}(\neg b, \rho) &= \neg \mathcal{B}_{abstr}(b, \rho) \\
\mathcal{B}_{abstr}(b_1 \wedge b_2, \rho) &= \mathcal{B}_{abstr}(b_1, \rho) \wedge \mathcal{B}_{abstr}(b_2, \rho) \\
\mathcal{B}_{abstr}(b_1 \vee b_2, \rho) &= \mathcal{B}_{abstr}(b_1, \rho) \vee \mathcal{B}_{abstr}(b_2, \rho)
\end{aligned}$$

Dabei ist zu beachten, dass die logischen Operationen auf der rechten Seite Operationen der dreiwertigen Logik sind.

Wir zeigen nun noch, dass die abstrakte Auswertung von Booleschen und arithmetischen Ausdrücken auch tatsächlich dem Gedanken der abstrakten Interpretation entspricht.

Satz 5.1.17 *Es sei a ein arithmetischer Ausdruck. Dann gilt $\beta(\mathcal{A}(a,\sigma)) \in \mathcal{A}_{abstr}(a, \beta \circ \sigma)$.*
Des weiteren sei b ein Boolescher Ausdruck. Dann gilt $\mathcal{B}(b,\sigma) \in val(\mathcal{B}_{abstr}(b, \beta \circ \sigma))$.

Beweis: Durch strukturelle Induktion über den Aufbau von a bzw. b. □

Ähnlich wie in Anhang A können wir nun mit Hilfe folgender Regeln die abstrakte Semantik von WHILE-Programmen definieren (siehe Tabelle 5.1). Dabei gilt $abs \in \mathbf{Abs}$, d.h. $abs \subseteq (\mathbf{Var} \to A)$.

$$\langle [\mathtt{x:=}a]^\ell, abs\rangle \Longrightarrow \{\rho[\mathtt{x} \mapsto w] \mid \rho \in abs, w \in \mathcal{A}_{abstr}(a,\rho)\} \quad \textit{[ass]}$$

$$\langle [\mathtt{skip}]^\ell, abs\rangle \Longrightarrow abs \quad \textit{[skip]}$$

$$\frac{\langle S_1, abs\rangle \Longrightarrow \langle S_1', abs'\rangle}{\langle S_1\,\mathtt{;}\,S_2, abs\rangle \Longrightarrow \langle S_1'\,\mathtt{;}\,S_2, abs'\rangle} \quad \textit{[seq1]}$$

$$\frac{\langle S_1, abs\rangle \Longrightarrow abs'}{\langle S_1\,\mathtt{;}\,S_2, abs\rangle \Longrightarrow \langle S_2, abs'\rangle} \quad \textit{[seq2]}$$

$$\langle \mathtt{if}\ [b]^\ell\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2\ \mathtt{fi}, abs\rangle \Longrightarrow \langle S_1, abs \backslash \{\rho \mid \mathcal{B}_{abstr}(b,\rho) = 0\}\rangle \quad \textit{[if1]}$$

$$\langle \mathtt{if}\ [b]^\ell\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2\ \mathtt{fi}, abs\rangle \Longrightarrow \langle S_2, abs \backslash \{\rho \mid \mathcal{B}_{abstr}(b,\rho) = 1\}\rangle \quad \textit{[if2]}$$

$$\langle \mathtt{while}\ [b]^\ell\ \mathtt{do}\ S\ \mathtt{od}, abs\rangle \Longrightarrow \langle S\,\mathtt{;}\,\mathtt{while}\ [b]^\ell\ \mathtt{do}\ S\ \mathtt{od}, abs \backslash \{\rho \mid \mathcal{B}_{abstr}(b,\rho) = 0\}\rangle \quad \textit{[wh1]}$$

$$\langle \mathtt{while}\ [b]^\ell\ \mathtt{do}\ S\ \mathtt{od}, abs\rangle \Longrightarrow abs \backslash \{\rho \mid \mathcal{B}_{abstr}(b,\rho) = 1\} \quad \textit{[wh2]}$$

Tabelle 5.1: Abstrakte Semantik von WHILE-Programmen

Es ist zu beachten, dass bei bedingten Anweisungen diejenigen abstrakten Werte entfernt werden, die auf jeden Fall die jeweils andere Verzweigung ausgelöst hätten (siehe *[if1]*, *[if2]*, *[wh1]*, *[wh2]*). Aufbauend auf dieser Semantik kann man nun auch die abstrakte $next^\#$-Funktion definieren:

Definition 5.1.18 *Wir definieren:*

$$next^\#_{S,S'}(abs) = \begin{cases} abs' & \textit{falls } \langle S, abs\rangle \Longrightarrow \langle S', abs'\rangle \\ \emptyset & \textit{sonst} \end{cases}$$

$$next^\#_{S,\downarrow}(abs) = \begin{cases} abs' & \textit{falls } \langle S, abs\rangle \Longrightarrow abs' \\ \emptyset & \textit{sonst} \end{cases}$$

Aufgabe 5.1.19 Zeigen Sie an einem Beispiel, dass die abstrakte Semantik aus Tabelle 5.1 im allgemeinen nicht die genaueste abstrakte Semantik eines WHILE-Programms ist.

Theorem 5.1.20 (Korrektheit der abstrakten Semantik) *Die in Definition 5.1.18 definierte Familie von $next^\#$-Funktion ist eine korrekte abstrakte Semantik im Sinne von Definition 5.1.13.*

Beweis: Wir müssen nun zeigen, dass

$$\alpha(next_{S,S'}(\gamma(abs))) \subseteq next^\#_{S,S'}(abs)$$

für $abs \in \mathbf{Abs}$ und alle WHILE-Programme S, S' und $S' = \downarrow$ gilt. Da $\langle \alpha, \gamma\rangle$ eine Galois-Verbindung ist und die Funktion $next^\#_{S,S'}$ monoton ist, folgt dies aus

$$\alpha(next_{S,S'}(\Sigma)) \subseteq next^\#_{S,S'}(\alpha(\Sigma))$$

für alle $\Sigma \subseteq \mathbf{State}$.

Sei nun $\rho' \in abs' = \alpha(next_{S,S'}(\Sigma)) \neq \emptyset$, d.h., es gibt ein $\sigma \in \Sigma$ mit $\langle S, \sigma\rangle \to \langle S', \sigma'\rangle$ (bzw. $\langle S, \sigma\rangle \to \sigma'$, falls $S' = \downarrow$) und es gilt $\rho' = \beta \circ \sigma'$. Wir müssen nun zeigen, dass ρ' in $next^{\#}_{S,S'}(\alpha(\Sigma))$ enhalten ist und machen dazu eine strukturelle Induktion über den Aufbau von S. Dabei müssen wir noch eine Fallunterscheidung bezüglich S' mit einbeziehen.

Falls S' nicht direkt von S aus erreichbar ist, so gilt $next_{S,S'}(\Sigma) = \emptyset$ und daraus folgt $\alpha(next_{S,S'}(\Sigma)) = \emptyset$ (nach Definition dieser speziellen Galois-Verbindung). Daraus folgt trivialerweise die Inklusion.

- Es gilt $S = [\texttt{x:=}a]^{\ell}$ und $S' = \downarrow$. In diesem Fall gilt $\sigma' = \sigma[\texttt{x} \mapsto \mathcal{A}(a, \sigma)]$. Es gilt $\beta \circ \sigma \in abs = \alpha(\Sigma)$ und mit der Definition der abstrakten Semantik ergibt sich $(\beta \circ \sigma)[\texttt{x} \mapsto \beta(\mathcal{A}(a, \sigma))] \in next^{\#}_{S,S'}(\alpha(\Sigma))$. Denn nach Satz 5.1.17 gilt $\beta(\mathcal{A}(a, \sigma)) \in \mathcal{A}_{abstr}(a, \beta \circ \sigma)$. Und wegen $\beta \circ \sigma' = (\beta \circ \sigma)[\texttt{x} \mapsto \beta(\mathcal{A}(a, \sigma))]$ erhalten wir $\rho' = \beta \circ \sigma' \in next^{\#}_{S,S'}(\alpha(\Sigma))$.

- Es gilt $S = [\texttt{skip}]^{\ell}$ und $S' = \downarrow$. Es gilt $\sigma' = \sigma$ und $\beta \circ \sigma \in \alpha(\Sigma)$. Mit der Definition der abstrakten Semantik folgt $\rho' = \beta \circ \sigma' = \beta \circ \sigma \in next^{\#}_{S,S'}(\alpha(\Sigma))$.

- Es gilt $S = S_1; S_2$ und $S' = S_1'; S_2$. Nach Induktionsvoraussetzung wissen wir bereits, dass $\alpha(next_{S_1,S_1'}(\Sigma)) \subseteq next^{\#}_{S_1,S_1'}(\alpha(\Sigma))$. Und wegen $next_{S_1;S_2,S_1';S_2}(\Sigma) = next_{S_1,S_1'}(\Sigma)$ und $next^{\#}_{S_1;S_2,S_1';S_2}(\alpha(\Sigma)) = next^{\#}_{S_1,S_1'}(\alpha(\Sigma))$ folgt dann $\alpha(next_{S,S'}(\Sigma)) \subseteq next^{\#}_{S,S'}(\alpha(\Sigma))$ und damit auch $\rho' \in next^{\#}_{S,S'}(\alpha(\Sigma))$.

- Es gilt $S = S_1; S_2$ und $S' = S_2$. Nach Induktionsvoraussetzung wissen wir bereits, dass $\alpha(next_{S_1,\downarrow}(\Sigma)) \subseteq next^{\#}_{S_1,\downarrow}(\alpha(\Sigma))$. Und wegen $next_{S_1;S_2,S_2}(\Sigma) = next_{S_1,\downarrow}(\Sigma)$ und $next^{\#}_{S_1;S_2,S_2}(\alpha(\Sigma)) = next^{\#}_{S_1,\downarrow}(\alpha(\Sigma))$ folgt dann $\alpha(next_{S,S'}(\Sigma)) \subseteq next^{\#}_{S,S'}(\alpha(\Sigma))$ und damit auch $\rho' \in next^{\#}_{S,S'}(\alpha(\Sigma))$.

- Es gilt $S = \texttt{if } [b]^{\ell} \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ fi}$ und $S' = S_1$. Es gilt $\sigma' = \sigma$ und $\beta \circ \sigma \in abs = \alpha(\Sigma)$. Aufgrund des Übergangs nach S_1 wissen wir, dass $\mathcal{B}(b, \sigma) = \mathit{true}$ gilt. Daraus folgt mit Satz 5.1.17, dass $\mathit{true} \in val(\mathcal{B}_{abstr}(b, \beta \circ \sigma))$ und damit $\mathcal{B}_{abstr}(b, \beta \circ \sigma) \neq 0$. Damit gehört $\beta \circ \sigma$ nicht zu den abstrakten Belegungen, die durch die Reduktionsregel aus abs entfernt werden und es gilt damit $\rho' = \beta \circ \sigma' = \beta \circ \sigma \in next^{\#}_{S,S'}(\alpha(\Sigma))$.

- Es gilt $S = \texttt{if } [b]^{\ell} \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ fi}$ und $S' = S_2$. Dies ist analog zum vorherigen Fall.

- Es gilt $S = \texttt{while } [b]^{\ell} \texttt{ do } P \texttt{ od}$ und $S' = P\texttt{;while } [b]^{\ell} \texttt{ do } P \texttt{ od}$. Außerdem ist $\sigma' = \sigma$ und $\beta \circ \sigma \in abs = \alpha(\Sigma)$. Aufgrund der Form des Übergangs wissen wir, dass $\mathcal{B}(b, \sigma) = \mathit{true}$ gilt. Daraus folgt mit Satz 5.1.17, dass $\mathit{true} \in val(\mathcal{B}_{abstr}(b, \beta \circ \sigma))$. Damit gehört $\beta \circ \sigma$ nicht zu den abstrakten Belegungen, die durch die Reduktionsregel aus abs entfernt werden und es gilt damit $\rho' = \beta \circ \sigma' = \beta \circ \sigma \in next^{\#}_{S,S'}(\alpha(\Sigma))$.

- Es gilt $S = \texttt{while } [b]^{\ell} \texttt{ do } P \texttt{ od}$ und $S' = \downarrow$. Außerdem ist $\sigma' = \sigma$ und $\beta \circ \sigma \in abs = \alpha(\Sigma)$. Aufgrund der Form des Übergangs wissen wir, dass $\mathcal{B}(b, \sigma) = \mathit{false}$ gilt. Daraus folgt mit Satz 5.1.17, dass $\mathit{false} \in val(\mathcal{B}_{abstr})(b, \beta \circ \sigma)$. Damit gehört $\beta \circ \sigma$ nicht zu den abstrakten Belegungen, die durch die Reduktionsregel aus abs entfernt werden und es gilt damit $\rho' = \beta \circ \sigma' = \beta \circ \sigma \in next^{\#}_{S,S'}(\alpha(\Sigma))$.

□

5.1.5 Anwendung: Verifikation von 16-Bit-Multiplikation

Wir wenden die oben vorgestellten Techniken an, um ein Programm zu verifizieren, das zwei 16-Bit-Zahlen miteinander multipliziert. Dieses Beispiel wird in [CGP00, CGL99] behandelt. Wir

werden die oben vorgestellten Abstraktionen geringfügig verallgemeiner, indem nun u.a. auch 16-Bit-Zahlen als Datentypen erlaubt sind.

Folgende Operationen werden in dem Multiplikationsprogramm verwendet: falls `z` eine n-Bit-Zahl darstellt und $m \geq n$ gilt, so beschreibt `(z:m)` eine m-Bit-Zahl, die dadurch entstanden ist, dass (unter Umständen) führende Nullen zu `x` hinzugefügt wurden. Falls `x` und `y` Variable sind, die für m- bzw. n-Bit-Zahlen stehen und `z` eine $m+n$-Bit-Zahl ist, so bezeichnet die Zuweisung `(x,y):= z` das Splitten von `z` in die ersten (höchstwertigen) m Bits und die restlichen (niedrigstwertigen) n Bits und die Zuweisung der jeweiligen Werte an `x` und `y`. Mit `lsb(z)` meint man das niedrigstwertige Bit von z (least significant bit). Mit `z>>1` bzw. `z<<1` bezeichnet man einen Shift um ein Bit nach rechts bzw. links.

Der Rest des Programms sollte selbsterklärend sein, `f1` und `f2` bezeichnen die miteinander zu multiplizierenden Faktoren, `out` ist das 16-Bit-Ergebnis und `overflow` bezeichnet das Overflow-Bit.

Programm 5.1.21 (Multiplikation)

```
[out := 0]1;
[overflow := 0]2;
while [(f1≠0) ∧ (overflow=0)]3 do
   if [lsb(f1)=1]4
      then [(overflow,out) := (out:17) + f2]5
      else [skip]6
   fi;
   [f1 := f1 >> 1]7;
   if [(f1≠0) ∧ (overflow=0)]8
      then [(overflow,f2) := (f2:17)<<1]9
      else [skip]10
   fi
od
```

In [CGL99] wurde dieses Programm auf abstrakten Werten interpretiert, wobei der zweite Faktor `f2` und die Ausgabe `out` modulo einer festen Zahl n genommen wurde. Das Flag `overflow` hat nur ein Bit und muss daher nicht abstrahiert werden und der erste Faktor `f1` kontrolliert die gesamte Multiplikation, so dass es bei einer Abstraktion dieser Variable zu sehr ungenauen Ergebnissen kommen würde. Die Variable `f1` wird daher in diesem Fall nicht abstrahiert. Wir kennzeichnen Binärzahlen durch ein nachgestelltes b.

Wir sehen uns an was für den Fall `f1 = 101b` $(= 5)$, `f2 = 1001010b` $(= 74)$ und $n = 5$ passiert. Es gilt $74 \equiv 4 \pmod 5$ und `out` und `overflow` sind zu Beginn undefiniert, wir beginnen daher mit dem abstrakten Wert $abs = \{[\texttt{f1} \mapsto 101b, \texttt{f2} \mapsto 4, \texttt{out} \mapsto m, \texttt{overflow} \mapsto b] \mid m \in \{0,1,2,3,4\}, b \in \{0,1\}\}$. Der Beginn des Ablaufs wird in Abbildung 5.5) dargestellt. Die Variable `overflow` wird dort mit `of` abgekürzt.

Da die beiden Variablen `f2` und `out` modulo 5 abstrahiert wurden, kann bei einer Addition prinzipiell immer ein Übertrag auftreten. Dies muss bei der abstrakten Semantik berücksichtigt werden. Außerdem entspricht der Shift nach links einer Multiplikation mit 2, was bei der Zahl 4 modulo fünf gerechnet den Wert 3 ergibt.

Man beachte, dass man bei Auftreten eines Overflows den Wert von `out` durchaus noch genauer als in Abbildung 5.5 angegeben, bestimmen kann. Sei n der abstrakte Wert von `f2` und m der abstrakte Wert von `out` (jeweils modulo 5), so ergibt sich bei Auftreten eines Overflows in Block 5 für den abstrakten Wert von `out`: $(m + n - 2^{16}) \mod 5 = (m + n - 1) \mod 5$. Das durch den Overflow aufgetretene 17-te Bit wird dabei entfernt.

Das Programm wurde für alle Eingaben `f1` und für Modulo-Rechnung mit $n = 5, 7, 9, 11, 32$ getestet und das Resultat war entweder ein gesetztes `overflow`-Flag oder das Multiplikationsergebnis war korrekt modulo n. Außerdem ist in diesem Fall die Menge der abstrakten Werte für `out` jeweils einelementig. Mit dem chinesischen Restsatz kann man sehen, dass damit das gesamte Programm korrekt ist.

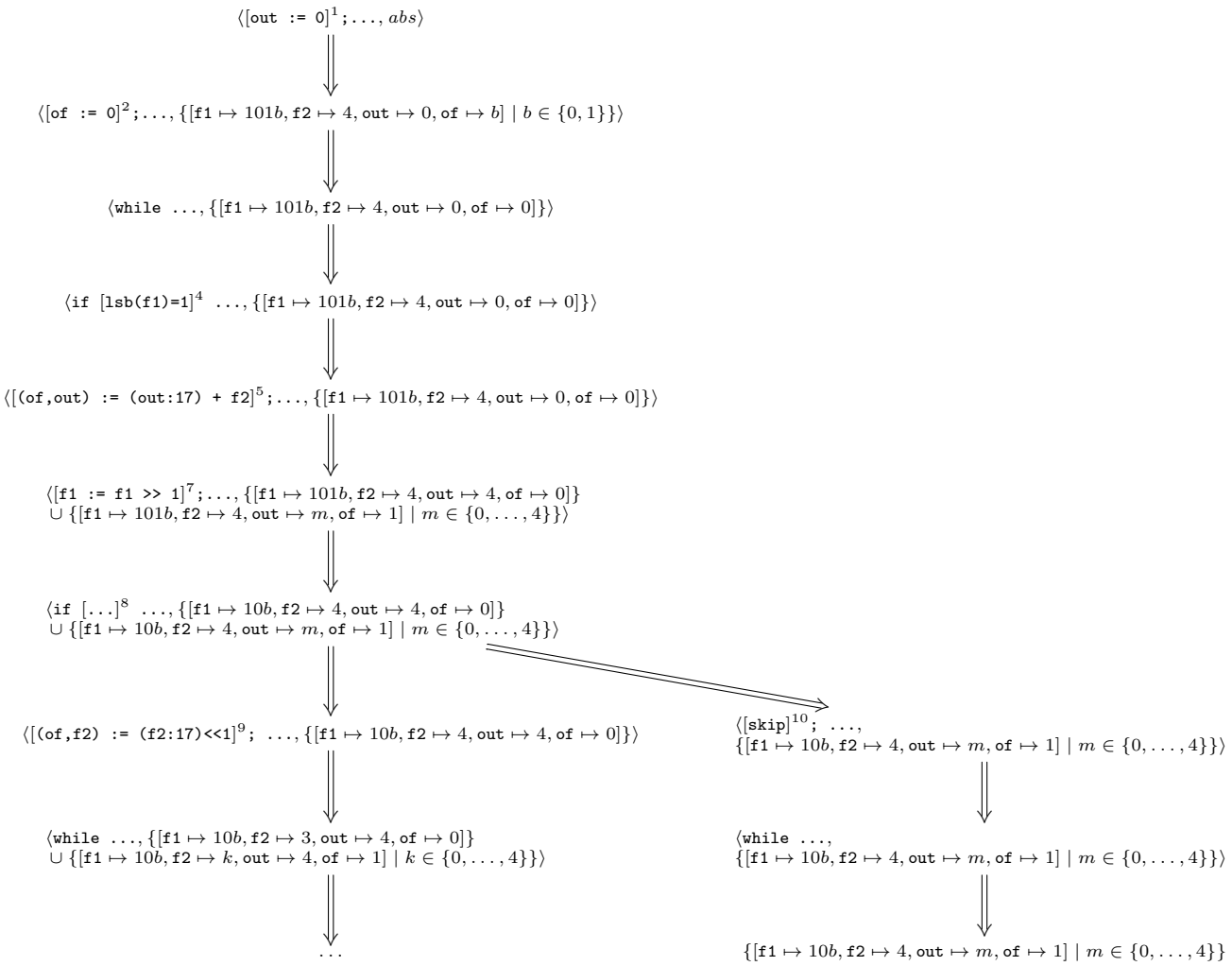


Abbildung 5.5: Übergänge des abstrakten Multiplikations-Programms

Satz 5.1.22 (Chinesischer Restsatz) *Seien $m_1, \ldots, m_k \in \mathbb{N}_0 \backslash \{0\}$, so dass gilt $ggT(m_i, m_j) = 1$ für $i, j \in \{1, \ldots, k\}$ und $i \neq j$, d.h., die Zahlen $m_1, \ldots, m_k$ sind paarweise teilerfremd. Sei außerdem $m = m_1 \cdots m_k$ und seien $z_1, \ldots, z_k \in \mathbb{Z}$.*

Dann gibt es ein eindeutiges $z \in \mathbb{Z}$, so dass gilt

$$0 \leq z < m \quad \textit{und} \quad z \equiv z_j \pmod{m_j} \textit{ für alle } j \in \{1, \ldots, k\}.$$

Hier entspricht z dem Ergebnis `out` und die z_j entsprechen den Modulo-Ergebnissen für `out` in der abstrakten Semantik, wobei wir ein festes `f1` und einen festen abstrakten Wert für `f2` annehmen. Außerdem gilt $m_1 = 5$, $m_2 = 7$, $m_3 = 9$, $m_4 = 11$ und $m_5 = 32$. Demnach gibt es ein eindeutiges `out` modulo $m = 5 \cdot 7 \cdot 9 \cdot 11 \cdot 32 = 110.880$, so dass $\texttt{out} \equiv z_j \pmod{m_j}$ gilt. Weil aber $m > 2^{16}$ gilt und außerdem die abstrakte Semantik korrekt ist, kann aufgrund der Eindeutigkeit der Lösung dieses `out` nur das korrekte Multiplikationsergebnis sein. Daraus folgt für die Korrektheit des Programms: entweder ist bei Terminierung das `overflow`-Flag gesetzt, oder die Multiplikation war korrekt.

Das vollständige Austesten des Programms ist bei 16-Bit-Zahlen zwar grundsätzlich möglich, war aber hier viel zu aufwändig. In diesem Fall müsste man $2^{16} \cdot 2^{16} = 2^{32} = 4,29 \cdot 10^9$ Möglichkeiten durchtesten, bei der Verifikation mit Hilfe abstrakter Interpretation sind es "nur" $2^{16} \cdot (5 + 7 + 9 + 11 + 32) = 4.194.304 = 4,19 \cdot 10^6$.

5.2 Prädikatabstraktion und Abstraktionsverfeinerung

Die Abstraktionen, die wir bisher betrachtet haben, haben im wesentlichen jede Variable "einzeln" abstrahiert, wodurch es sehr schwierig ist, Aussagen darüber zu treffen, wie sich die Variablenwerte zueinander verhalten. Fragen wie beispielsweise: "Gilt an einer bestimmten Programmstelle `x<y`?" können dadurch kaum beantwortet werden. Dieses Problem wird durch Einführung der sogenannten*Prädikatabstraktion* behoben, die es erlaubt, Variablenbelegungen danach zu klassifizieren, ob bestimmte Prädikate (wie beispielsweise `x<y`) erfüllt sind. Dadurch ist es dann möglich, eine unendliche Menge von Variablenbelegungen durch endlich viele abstrakte Werte zu approximieren.

Nun stellt sich allerdings noch die Frage, wie man die Prädikate geeignet wählt. Diese müssen entweder vom Benutzer eingegeben werden, was schwierig und mühsam sein kann, oder aus dem zu verifizierenden Programm extrahiert werden. Eine Methode, die letzteres liefert, ist die sogenannte *Abstraktionsverfeinerung basierend auf Gegenbeispielen* (counterexample-guided abstraction refinement). Hierbei wird, ausgehend von einer groben Abstraktion, zunächst ein Gegenbeispiel gesucht, d.h., ein Ablauf der zu einem fehlerhaften Zustand führt. Daraufhin wird geprüft, ob dieser Ablauf echt ist (in diesem Fall ist das Programm wirklich fehlerhaft), oder ob er unecht ist, d.h., er wurde erst durch die Approximation eingeführt. Im letzteren Fall kann man dann die Abstraktion durch Hinzufügen weiterer Prädikate so verfeinern, dass zumindest das gefundene Gegenbeispiel verschwindet. Welche Prädikate gewählt werden, ergibt sich durch Untersuchung des Gegenbeispiels.

5.2.1 Hoare-Logik

Um Prädikatabstraktion verstehen zu können, ist es nützlich, sich zuerst mit Hoare-Logik zu beschäftigen (siehe [Apt81, Dij75]). Dabei sind ein Block C einer Programmiersprache (bei uns: WHILE-Programme) und zwei Prädikate p, q über den Programmvariablen gegeben.[1] Man schreibt $\{p\}\, C\, \{q\}$ (genannt: Hoare-Tripel), wenn sichergestellt ist, dass nach Ausführung von C das Prädikat q erfüllt ist, wenn vor Ausführung von C das Prädikat p erfüllt ist. Das Prädikat p heißt dabei *Vorbedingung (precondition)*, das Prädikat q *Nachbedingung (postcondition)*. Außerdem kann C in unserem Fall eine Zuweisung, ein Skip-Kommando oder eine Bedingung sein. Falls $C = [b]^\ell$ ein Bedingungsblock ist, so dass b unter p nicht erfüllt ist, so ist nichts zu zeigen und q kann beliebig sein.

[1] Man sagt statt Prädikat auch Formel oder Boolescher Ausdruck.

Beispiel 5.2.1 Folgende Kombinationen von Kommandos und Prädikaten sind gültige Hoare-Tripel:

- $\{\texttt{x=7}\}\,[\texttt{x:=x+2}]^\ell\,\{\texttt{x=9}\}$
- $\{\texttt{x=7}\}\,[\texttt{x:=x+2}]^\ell\,\{\mathit{true}\}$
- $\{\texttt{x<y}\}\,[\texttt{y:=y+1}]^\ell\,\{\texttt{x<y}\}$
- $\{\texttt{x<y}\}\,[\texttt{y:=y+1}]^\ell\,\{\texttt{x+1<y}\}$
- $\{\texttt{x<y}\}\,[\texttt{skip}]^\ell\,\{\texttt{x<y}\}$
- $\{\texttt{x+y=y}\}\,[\texttt{x:=x+y}]^\ell\,\{\texttt{x=y}\}$
- $\{\texttt{x=0}\}\,[\texttt{x:=x+y}]^\ell\,\{\texttt{x=y}\}$
- $\{\texttt{x=y}\}\,[\texttt{x:=x+y}]^\ell\,\{\texttt{x-y=y}\}$
- $\{\texttt{x=y}\}\,[\texttt{x:=x+y}]^\ell\,\{\texttt{x=2y}\}$
- $\{\texttt{x}\leq\texttt{y}\}\,[\texttt{x}\geq\texttt{y}]^\ell\,\{\texttt{x=y}\}$

Als Prädikate erlauben wir Boolesche Ausdrücke wie sie in Anhang A.1 definiert sind mit der entsprechenden Auswertungsfunktion $\mathcal{B}$: **BExp** $\times$ **State** $\to \{\mathit{true}, \mathit{false}\}$, d.h., wir betrachten wieder eine zweiwertige Logik. Zusätzlich erlauben wir Quantifizierung über Variablen, sowohl Allquantoren ($\forall\texttt{x}$), als auch Existenzquantoren ($\exists\texttt{x}$). Dabei wird die induktive Definition der Auswertungsfunktion folgendermaßen auf Quantoren erweitert:

Allquantor: $\mathcal{B}(\forall\texttt{x}\,p,\sigma) = \begin{cases} \mathit{true} & \text{falls für alle } z\in\mathbb{Z} \text{ gilt: } \mathcal{B}(p,\sigma[\texttt{x}\mapsto z]) = \mathit{true} \\ \mathit{false} & \text{sonst} \end{cases}$

Existenzquantor: $\mathcal{B}(\exists\texttt{x}\,p,\sigma) = \begin{cases} \mathit{true} & \text{falls es ein } z\in\mathbb{Z} \text{ gibt mit } \mathcal{B}(p,\sigma[\texttt{x}\mapsto z]) = \mathit{true} \\ \mathit{false} & \text{sonst} \end{cases}$

In Anlehnung an die Prädikatenlogik 1. Stufe schreiben wir auch $\sigma \models b$, falls $\mathcal{B}(b,\sigma) = \mathit{true}$ und sagen, dass σ ein Modell für b ist. Im Gegensatz zur Prädikatenlogik ist jedoch der Begriff einer Struktur bzw. eines Modells sehr stark eingeschränkt. Wir haben ein festes Universum (nämlich $\mathbb{Z}$) und die Interpretation aller Prädikate und Funktionssymbole ist festgelegt. Der einzige Freiheitsgrad ist noch die Festlegung der freien Variablen durch die Variablenbelegung σ.

Dennoch kann man einige nützliche Konzepte der Prädikatenlogik übernehmen. Insbesondere sagen wir, dass q eine *Folgerung* von p ist (in Zeichen: $p \models q$), wenn für alle $\sigma \in$ **State** aus $\sigma \models p$ immer $\sigma \models q$ folgt. In diesem Fall sagt man auch: p ist *stärker* als q bzw. q ist *schwächer* als p.

Falls sowohl $p \models q$, als auch $q \models p$ gilt, dann schreiben wir $p \equiv q$ und sagen, dass p und q *äquivalent* sind.

Wir kommen nun wieder zu den Hoare-Tripeln zurück und beschreiben die Anforderungen, die an diese gestellt werden:

Zuweisung: Falls $\{p\}\,[\texttt{x:=}a]^\ell\,\{q\}$, $\sigma \models p$ und $\langle[\texttt{x:=}a]^\ell,\sigma\rangle \to \sigma'$, dann gilt $\sigma' \models q$.

Skip-Anweisung: Falls $\{p\}\,[\texttt{skip}]^\ell\,\{q\}$, $\sigma \models p$ und $\langle[\texttt{skip}]^\ell,\sigma\rangle \to \sigma'$, dann gilt $\sigma' \models q$.

If-Then-Else-Bedingung (Then-Fall):

Falls $\{p\}\,[b]^\ell\,\{q\}$, $\sigma \models p$ und $\langle \texttt{if } [b]^\ell \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ fi}, \sigma\rangle \to \langle S_1, \sigma'\rangle$, dann gilt $\sigma' \models q$.

If-Then-Else-Bedingung (Else-Fall):

Falls $\{p\}\,[\neg b]^\ell\,\{q\}$, $\sigma \models p$ und $\langle \texttt{if } [b]^\ell \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ fi}, \sigma\rangle \to \langle S_2, \sigma'\rangle$, dann gilt $\sigma' \models q$.

While-Schleife (Eintritt in die Schleife):

Falls $\{p\}\,[b]^\ell\,\{q\}$, $\sigma \models p$ und $\langle \texttt{while } [b]^\ell \texttt{ do } S \texttt{ od}, \sigma\rangle \to \langle S\texttt{;while } [b]^\ell \texttt{ do } S \texttt{ od}, \sigma'\rangle$, dann gilt $\sigma' \models q$.

While-Schleife (Terminierung):

Falls $\{p\}\,[\neg b]^\ell\,\{q\}$, $\sigma \models p$ und $\langle \texttt{while } [b]^\ell \texttt{ do } S \texttt{ od}, \sigma\rangle \to \sigma'$, dann gilt $\sigma' \models q$.

Man beachte, dass beim Else-Fall der If-Then-Else-Anweisung bzw. bei der Terminierung der While-Schleife das Hoare-Tripel jeweils für die negierte Bedingung gebildet wird.

Für eine sequentielle Komposition $C_1; \ldots; C_n$ von Blöcken (entsprechend einem Pfad im Flussgraphen) schreiben wir $\{p\}\, C_1; \ldots; C_n\, \{q\}$, falls es Prädikate $p_1, \ldots, p_{n-1}$ gibt mit

$$\{p\}\, C_1\, \{p_1\} \quad \{p_1\}\, C_2\, \{p_2\} \quad \ldots \quad \{p_{n-2}\}\, C_{n-1}\, \{p_{n-1}\} \quad \{p_{n-1}\}\, C_n\, \{q\}$$

Hoare-Logik kann auch Aussagen über ganze Programme (inklusive While-Schleifen) machen, in dieser Allgemeinheit benötigen wir sie hier jedoch nicht. Es reicht aus, wenn wir Vor- und Nachbedingungen für Abläufe, d.h., für Sequenzen von Blöcken darstellen können.

Im nächsten Schritt wenden wir uns der Aufgabe zu, für ein gegebenes q und C ein geeignetes p zu finden mit $\{p\}\, C\, \{q\}$. Wenn man keine weiteren Anforderungen an p stellt, so ist diese Aufgabe trivial, man wählt einfach $p = \mathit{false}$. Da für kein σ die Bedingung $\sigma \models p$ erfüllt ist, haben wir daher auch keine Beweisverpflichtung. Eine kompliziertere Aufgabe ist es, die *schwächste Vorbedingung* p zu finden, für die $\{p\}\, C\, \{q\}$ gilt. Ebenso suchen wir nach der *stärksten Nachbedingung*.

Satz 5.2.2 (Schwächste Vorbedingung) *Gegeben sei ein Prädikat q und ein Block C. Dann bezeichnen wir die* schwächste Vorbedingung (weakest precondition) *p, für die $\{p\}\, C\, \{q\}$ gilt mit $wp(C, q)$. Sie kann folgendermaßen ermittelt werden:*

Zuweisung: $wp([\mathtt{x:=}a]^\ell, q) = q[\mathtt{x}/a]$

(Dabei bezeichnet $q[\mathtt{x}/a]$ die Bedingung q, in der `x` durch a ersetzt wurde.)

Skip-Anweisung: $wp([\mathtt{skip}]^\ell, q) = q$

Bedingung: $wp([b]^\ell, q) = (q \wedge b) \vee \neg b \equiv q \vee \neg b \equiv b \rightarrow q$

Das heißt bei einer Zuweisung wird gefordert, dass vorher q gilt, wobei aber für die Variable `x` (der der neue Wert des arithmetischen Ausdrucks a noch nicht zugewiesen wurde) a eingesetzt wird.

Bei einer Bedingung b wird entweder gefordert, dass q und die Bedingung selbst erfüllt sind, oder dass b nicht erfüllt ist. In letzterem Fall muss q nicht gelten, da es dann keine Möglichkeit gibt, die Bedingung entsprechend zu "überqueren".

Satz 5.2.3 (Stärkste Nachbedingung) *Gegeben sei ein Prädikat p und ein Block C. Dann bezeichnen wir die* stärkste Nachbedingung (strongest postcondition) *q, für die $\{p\}\, C\, \{q\}$ gilt mit $sp(p, C)$. Sie kann folgendermaßen ermittelt werden:*

Zuweisung: $sp(p, [\mathtt{x:=}a]^\ell) = \exists \mathtt{x'}\, (p[\mathtt{x}/\mathtt{x'}] \wedge \mathtt{x} = \mathtt{a}[\mathtt{x}/\mathtt{x'}])$

Skip-Anweisung: $sp(p, [\mathtt{skip}]^\ell) = p$

Bedingung: $sp(p, [b]^\ell) = p \wedge b$

Bei einer Zuweisung wird angenommen, dass es einen früheren Wert für `x` gibt der mit `x'` bezeichnet wird. Für dieses `x'` muss immer noch die Vorbedingung p gelten. Der neue Wert für `x` ist dann der Wert des arithmetischen Ausdrucks a, wobei für die Zuweisung allerdings noch der alte Wert `x'` verwendet wird.

Bei einer Bedingung wird einfach ausgesagt, dass nach dem Kommando sowohl die Vorbedingung p als auch die Bedingung b selbst gelten müssen.

Die Bildung von schwächsten Vor- bzw. stärksten Nachbedingungen kann auch auf Sequenzen von Blöcken ausgedehnt werden, indem die obigen Bildungsgesetze iteriert angewandt werden. D.h., man definiert:

$$\begin{aligned} wp(C_1; \ldots; C_n, q) &= wp(C_1; \ldots; C_{n-1}, wp(C_n, q)) \\ sp(p, C_1; \ldots; C_n) &= sp(sp(p, C_1), C_2; \ldots; C_n) \end{aligned}$$

Man beachte außerdem dass die Bildung der schwächsten Vorbedingung etwas einfacher zu handhaben ist, da dabei kein Quantor eingeführt werden muss.

Aufgabe 5.2.4 Betrachten Sie folgende Folgen von Blöcken und bestimmen Sie jeweils die schwächste Vorbedingung für $q =$ `(x<y)` und die stärkste Nachbedingung für $p =$ `(x=3)`.

(a) $[\texttt{y<z}]^1;[\texttt{x:=z+1}]^2;[\texttt{z=3}]^3$

(b) $[\texttt{x<y-1}]^1;[\texttt{skip}]^2;[\texttt{x:=x+1}]^3$

Das Hoare-Tripel $\{\mathit{true}\}\, C_1;\ldots;C_n\, \{\mathit{false}\}$ bedeutet, dass der Ablauf $C_1;\ldots;C_n$ unter keiner Variablenbelegung durchführbar ist. Die Begründung dafür ist folgende: angenommen es gäbe eine Variablenbelegung σ, mit der dieser Ablauf möglich ist, so dass σ sich während des Ablaufs zu σ' verändert. Dann müsste aus $\sigma \models \mathit{true}$ (was immer gilt) folgen, dass $\sigma' \models \mathit{false}$ gilt, was aber niemals erfüllt ist. So erhalten wir einen Widerspruch.

Beispiel 5.2.5 Folgende Abläufe sind nicht durchführbar.

(a) $[\texttt{x=1}]^1;[\texttt{x=2}]^2$

(b) $[\texttt{x<y}]^1;[\texttt{x:=x+1}]^2;[\texttt{x>y}]^3$

(c) $[\texttt{x:=c}]^1;[\texttt{c:=c+1}]^2;[\texttt{y:=c}]^3;[\texttt{x=m}]^4;[\texttt{y=m}]^5$

5.2.2 Prädikatabstraktion

Wir gehen nun genauer auf die sogenannte Prädikatabstraktion, im Englischen auch *predicate abstraction* genannt. Geeignete Literatur hierzu ist [GS97, HJMM04].

Die Grundidee bei der Prädikatabstraktion ist, dass eine Menge von Prädikaten oder Booleschen Ausdrücken über den Variablen ausgewählt werden. Ein abstrakter Wert legt dann für jedes dieser Prädikate fest, ob es erfüllt ist, nicht erfüllt ist, oder keine Aussage getroffen werden kann. Dies kann auch als Konjunktion von negierten bzw. nicht negierten Prädikaten dargestellt werden.

Definition 5.2.6 (Prädikatabstraktion) *Sei* **Var** *eine Menge von Variablen und* $\{p_1,\ldots,p_n\}$ *eine Menge von Prädikaten über diesen Variablen. Wir betrachten nun den Verband* $\mathbf{Abs}(p_1,\ldots,p_n)$*, der wie folgt definiert ist:*

- *Die dem Verband zugrundeliegende Menge besteht aus:*
  - *allen Konjunktionen der Prädikate* $p_1,\ldots,p_n,\neg p_1,\ldots,\neg p_n$*, wobei aber kein Prädikat sowohl positiv als auch negativ vorkommt,*
  - *der Konstanten true (entspricht der leeren Konjunktion)* und
  - *der Konstanten false (entspricht einer Konjunktion der Form* $p_i \wedge \neg p_i$*).*

  Dabei werden äquivalente Formeln $p \equiv q$ *als gleich betrachtet.*
- *Die Verbandsordnung ist die Folgerungsrelation* $\models$*.*

Damit ist *false* das kleinste Element des Verbandes und *true* das größte. Zwei Formeln werden als gleich betrachtet, wenn sie äquivalent bezüglich $\equiv$ sind. D.h., eigentlich wird die Menge der oben beschriebenen Konjunktionen durch $\equiv$ faktorisiert.

Das Infimum entspricht der Konjunktion (falls dabei zwei Prädikate der Form p und $\neg p$ zusammentreffen, so erhält man *false*). Das Supremum von zwei Formeln q_1, q_2 sollte idealerweise mit Hilfe der Disjunktion $q_1 \vee q_2$ zu erhalten sein, jedoch ergibt sich dabei nicht immer eine Formel aus $\mathbf{Abs}(p_1,\ldots,p_n)$. Daher ergibt sich als Supremum die stärkste Formel q obiger Form, für die $q_1 \vee q_2 \models q$ gilt.

Sei q eine beliebige Formel. Im folgenden schreiben wir $\overline{q}$ für die stärkste Formel aus $\mathbf{Abs}(p_1,\ldots,p_n)$ für die $q \models \overline{q}$ gilt. Diese ist darstellbar als $\bigwedge\{q' \in \mathbf{Abs}(p_1,\ldots,p_n) \mid q \models q'\}$ (das ist auch wieder eine Konjunktion) und damit eindeutig bestimmt.

Beispiel 5.2.7 Wir betrachten drei Prädikate p_1, p_2, p_3.

(a) Gegeben seien die Formeln $q_1 = p_1 \wedge \neg p_2$, $q_2 = \neg p_2 \wedge p_3$. Dann gilt:

$$\begin{aligned} q_1 \sqcap q_2 &= q_1 \wedge q_2 \equiv p_1 \wedge \neg p_2 \wedge p_3 \\ q_1 \sqcup q_2 &= \overline{\overline{q_1} \vee \overline{q_2}} \equiv \overline{(p_1 \wedge \neg p_2) \vee (\neg p_2 \wedge p_3)} \equiv \overline{\neg p_2 \wedge (p_1 \vee p_3)} \end{aligned}$$

Es gilt $\overline{\neg p_2 \wedge (p_1 \vee p_3)} \models \neg p_2$, die Formeln sind jedoch nicht notwendigerweise äquivalent.

(b) Gegeben seien die Formeln $q_1 = p_1 \wedge \neg p_2$, $q_2 = p_1 \wedge p_2$. Dann gilt:

$$\begin{aligned} q_1 \sqcap q_2 &= q_1 \wedge q_2 \equiv \mathit{false} \\ q_1 \sqcup q_2 &= \overline{\overline{q_1} \vee \overline{q_2}} \equiv \overline{(p_1 \wedge \neg p_2) \vee (p_1 \wedge p_2)} \equiv \overline{p_1 \wedge (\neg p_2 \vee p_2)} \equiv \overline{\overline{p_1}} \equiv p_1 \end{aligned}$$

Damit können wir nun auch festlegen, wie die Galoisverbindung für die Prädikatabstraktion auszusehen hat.

Definition 5.2.8 (Galoisverbindung für die Prädikatabstraktion) *Sei* **Var** *eine Menge von Variablen und* $\{p_1, \ldots, p_n\}$ *eine Menge von Prädikaten über diesen Variablen. Wir definieren die* Galoisverbindung für die Prädikatabstraktion *wie folgt:*

$$\alpha \colon \mathcal{P}(\mathbf{State}) \to \mathbf{Abs}(p_1, \ldots, p_n) \qquad \gamma \colon \mathbf{Abs}(p_1, \ldots, p_n) \to \mathcal{P}(\mathbf{State})$$

Sei nun $q_\sigma \in \mathbf{Abs}(p_1, \ldots, p_n)$ *die stärkste Formel für die* $\sigma \models q$ *gilt, d.h.,* q_σ *enthält* $p_i \in \{p_1, \ldots, p_n\}$*, falls* $\sigma \models p_i$ *gilt und* $\neg p_i$*, falls* $\sigma \not\models p_i$ *gilt. Dann definieren wir:*

$$\alpha(\Sigma) = \bigsqcup \{q_\sigma \mid \sigma \in \Sigma\} \qquad \gamma(q) = \{\sigma \mid \sigma \models q\}$$

Damit ist bereits alles festgelegt und nun geht es noch darum, sich zu überlegen, wie die genaueste abstrakte Semantik unter diesen Festlegungen aussehen muss. Dabei hilft uns die weiter oben definierte Hoare-Logik und der Begriff der stärksten Nachbedingung. Angenommen, wir haben vor einem Block C den abstrakten Wert p aus $\mathbf{Abs}(p_1, \ldots, p_n)$. Nun wollen wir das stärkste Prädikat q aus $\mathbf{Abs}(p_1, \ldots, p_n)$ bestimmen, für das $\{p\}\, C\, \{q\}$ gilt. Dies läßt sich mit der bisher eingeführten Notation schreiben als $q = \overline{sp(p, C)}$.

Damit ergibt sich die in Tabelle 5.2 dargestellte abstrakte Semantik für die Prädikatabstraktion.

$$\langle [\texttt{x:=}a]^\ell, p\rangle \Longrightarrow \overline{sp(p, [\texttt{x:=}a]^\ell)} \;\; \textit{[ass]} \qquad \langle [\texttt{skip}]^\ell, p\rangle \Longrightarrow p \;\; \textit{[skip]}$$

$$\frac{\langle S_1, p\rangle \Longrightarrow \langle S_1', q\rangle}{\langle S_1\,\texttt{;}\,S_2, p\rangle \Longrightarrow \langle S_1'\,\texttt{;}\,S_2, q\rangle} \;\; \textit{[seq1]} \qquad \frac{\langle S_1, p\rangle \Longrightarrow q}{\langle S_1\,\texttt{;}\,S_2, p\rangle \Longrightarrow \langle S_2, q\rangle} \;\; \textit{[seq2]}$$

$$\langle \texttt{if}\ [b]^\ell\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{fi}, p\rangle \Longrightarrow \langle S_1, \overline{sp(p, [b]^\ell)}\rangle \;\; \textit{[if1]}$$

$$\langle \texttt{if}\ [b]^\ell\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{fi}, p\rangle \Longrightarrow \langle S_2, \overline{sp(p, [\neg b]^\ell)}\rangle \;\; \textit{[if2]}$$

$$\langle \texttt{while}\ [b]^\ell\ \texttt{do}\ S\ \texttt{od}, p\rangle \Longrightarrow \langle S\,\texttt{;while}\ [b]^\ell\ \texttt{do}\ S\ \texttt{od}, \overline{sp(p, [b]^\ell)}\rangle \;\; \textit{[wh1]}$$

$$\langle \texttt{while}\ [b]^\ell\ \texttt{do}\ S\ \texttt{od}, p\rangle \Longrightarrow \overline{sp(p, [\neg b]^\ell)} \;\; \textit{[wh2]}$$

Tabelle 5.2: Abstrakte Semantik bei der Prädikatabstraktion

Bemerkung: Ein Tupel $\langle S, \mathit{false}\rangle$ gilt als nicht erreichbar, denn es gibt keine Variablenbelegung σ, die *false* erfüllt, d.h., für die $\sigma \models \mathit{false}$ gilt.

Beispiel 5.2.9 Wir betrachten folgendes Programm:

```
if [x>y]^1 then
   while [y≠0]^2 do
      [x:=x-1]^3;[y:=y-1]^4
   od;
   if [x>y]^5
      then [skip]^6
      else [skip]^7
   fi
else [skip]^8 fi
```

Die While-Schleife wird nur betreten, wenn `(x>y)` gilt. Außerdem werden die Variablen `x` und `y` immer gleichzeitig dekrementiert. Das bedeutet, dass `(x>y)` eine Invariante ist, daher die Bedingung in Block 5 immer erfüllt sein muss und Block 7 nie betreten werden kann.

Wir weisen dies nach, indem wir Prädikatabstraktion mit den Prädikaten $p_1 =$ `(x>y)` und $p_2 =$ `(x≥y)` durchführen. Dadurch ergibt sich das in Abbildung 5.6 dargestellte abstrakte Transitionssystem. Abstrakte Zustände der Form $\langle S, \mathit{false}\rangle$ bzw. *false* sind nicht dargestellt. Daher ist aus Abbildung 5.6 abzulesen, dass Block 7 nicht erreichbar ist.

Zu beachten ist, dass Prädikatabstraktion allein mit dem Prädikat $p_1 =$ `(x>y)` nicht zum Erfolg geführt hätte, da nicht klar ist, ob dieses nach Block 3 noch gilt.

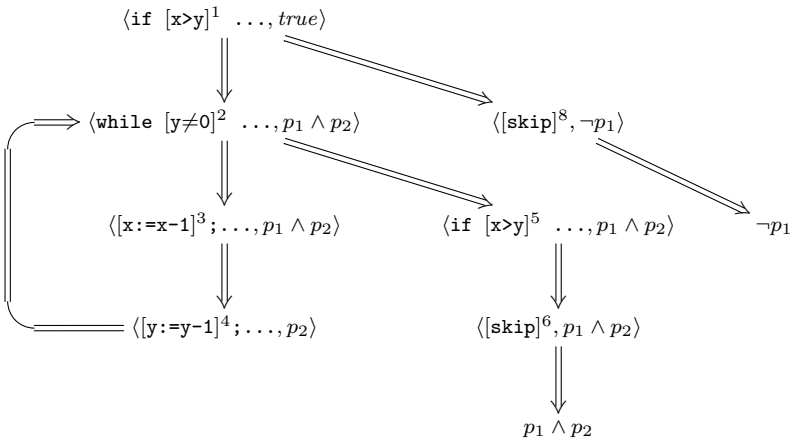


Abbildung 5.6: Beispiel für Prädikatabstraktion

Es gibt jedoch noch ein verstecktes Problem bei der Berechnung des abstrakten Transitionssystems: das Verbandselement $\overline{sp(p, C)}$ ist im allgemeinen nicht berechenbar. Das hängt mit Unentscheidbarkeitsresultaten für die Folgerungsbeziehung bei bestimmten Logiken zusammen. Für dieses Dilemma gibt es mehrere Lösungsmöglichkeiten:

- *Die Standardlösung: Über-Approximation.* Das heißt, man gibt sich auch mit schwächeren Nachbedingungen zufrieden. Diese erhält man in der Praxis im allgemeinen dadurch, dass man einen (automatischen) Theorembeweiser verwendet und damit versucht, für jedes $p_i \in \{p_1, \ldots, p_n\}$ die Folgerungsbeziehungen $sp(p, C) \models p_i$ und $sp(p, C) \models \neg p_i$ zu beweisen. Manchmal wird dies gelingen, manchmal muss die (evtl. nicht-terminierende) Berechnung durch einen Timeout abgebrochen werden. Als Approximation für $\overline{sp(p, C)}$ verwendet man dann die Konjunktion aller Literale, für die der Beweis gelungen ist (siehe auch [GS97]).

- *Wir betrachten nur bestimmte Typen von Programmen.* Es gibt Logiken über den ganzen Zahlen, für die die Folgerung $\models$ entscheidbar ist. Die bekannteste davon ist die sogenannte Presburger-Arithmetik, die jedoch nicht in der Lage ist, über Multiplikation zu sprechen. Das

heißt, hat man ein Programm, das nur Addition, aber keine Multiplikation enthält, so kann man $\overline{sp(p, C)}$ tatsächlich berechnen (siehe auch [HJMM04]).

- *Wir betrachten nur endliche Wertebereiche.* Wir nehmen an, dass unser Programm nicht auf allen ganzen Zahlen, sondern nur auf einem endlichen Wertebereich arbeitet, z.B., auf n-stelligen Binärzahlen. Das ist für Computer oft eine durchaus sinnvolle Einschränkung. Damit ist alles (der Wertebereich, die Galois-Verbindung, etc.) endlich und exakt bestimmbar. In der Praxis muss man jedoch geeignete Methoden finden, um möglicherweise exponentiell große Mengen von Variablenwerten kompakt darstellen zu können. Dazu werden oft Binary Decision Diagrams (BDDs) [And98] benutzt (siehe auch [EKS06]).

5.2.3 Abstraktionsverfeinerung

Wir beschäftigen uns nun mit der sogenannten *Abstraktionsverfeinerung basierend auf Gegebenbeispielen*, im Englischen *counterexample-guided abstraction refinement (CEGAR)* genannt.

Dabei soll eine bestimmte Eigenschaft P für ein Programm verifiziert werden. Diese Eigenschaft ist nicht weiter festgelegt, sie könnte aber beispielsweise folgendermaßen aussehen:

- Bestimmte Blöcke des Programms sind von keinem Ablauf erreichbar.
- Es erfolgt niemals eine Division durch 0.
- Der Wert der Variablen `x` wird niemals negativ.
- Bei Terminierung enthält `y` immer einen gerade Wert.

In Verbindung mit Model-Checking können auch komplexere Bedingung gestellt werden (siehe auch [CGJ+03]).

Es soll nun überprüft werden, ob das gegebene Programm S die Eigenschaft P erfüllt. Dazu wird zunächst eine (zumeist grobe) initiale Abstraktion von S erstellt, beispielsweise mit Hilfe der leeren Prädikatmenge. Dann wird überprüft wie man aus dieser groben Abstraktion bereits die Eigenschaft P ableiten kann. Wenn dies der Fall ist, dann war die Verifikation erfolgreich. Im allgemeinen wird das jedoch noch nicht der Fall sein und es gibt ein Gegenbeispiel, d.h., einen Ablauf, der P (möglicherweise) verletzt. Es wird nun überprüft, ob dieser Ablauf im ursprünglichen System möglich ist. Falls dies der Fall ist, so hat man einen Fehler gefunden. Falls nicht, d.h., der Ablauf ist unecht *(spurious counterexample)* so wird die Abstraktion so verfeinert, dass das unechte Gegenbeispiel verschwindet. Hier passiert das dadurch, dass weitere Prädikate zur Abstraktion hinzugefügt werden. Dies wird iteriert *(abstraction refinement loop)*, solange bis die Verifikation erfolgreich ist, ein Fehler gefunden wurde oder mit einem Timeout abgebrochen wird. Es gibt keine Garantie, dass das Verfahren immer terminiert.

Dieser Ablauf ist schematisch in Abbildung 5.7 dargestellt. Zu beachten ist, dass alle Schritte automatisch, d.h., ohne Unterstützung des Benutzers ablaufen. Diese Methode ist in mehreren existierenden Programmanalyse-Werkzeugen implementiert [CCG+03, BR01, HJMS02]. Weitere Artikel zum Thema sind [CGJ+03, McM03, HJMM04, EKS06].

Die hauptsächliche Frage ist nun: wie bestimmt man, ob ein Ablauf unecht ist und wie erhält man die neuen Prädikate, die zur Abstraktionsverfeinerung benötigt werden? Hier kommt uns wieder die Hoare-Logik zu Hilfe: angenommen wir betrachten den Ablauf $r = C_1; \ldots; C_n$, wobei Bedingungsblöcke negiert betrachtet werden, wenn der Else-Zweig gewählt wird bzw. die While-Schleife verlassen wird. Nach den Überlegungen in Abschnitt 5.2.1 ist er unecht genau dann, wenn $\{true\}\, r\, \{false\}$ gilt. Dies kann man dadurch ermitteln, indem man entweder $wp(r, false) \equiv true$ oder $sp(true, r) \equiv false$ überprüft.[2]

Es gibt dann Prädikate $p_1, \ldots, p_n$ mit

$$\{true\}\, C_1\, \{p_1\} \quad \{p_1\}\, C_2\, \{p_2\} \quad \ldots \quad \{p_{n-2}\}\, C_{n-1}\, \{p_{n-1}\} \quad \{p_{n-1}\}\, C_n\, \{false\},$$

[2]Dabei ist wiederum zu beachten, dass die Folgerung und damit auch die Äquivalenz nicht immer entscheidbar sind. Dieses Unentscheidbarkeitsproblem muss auf eine der Arten gelöst werden, die am Ende von Abschnitt 5.2.2 beschrieben sind.

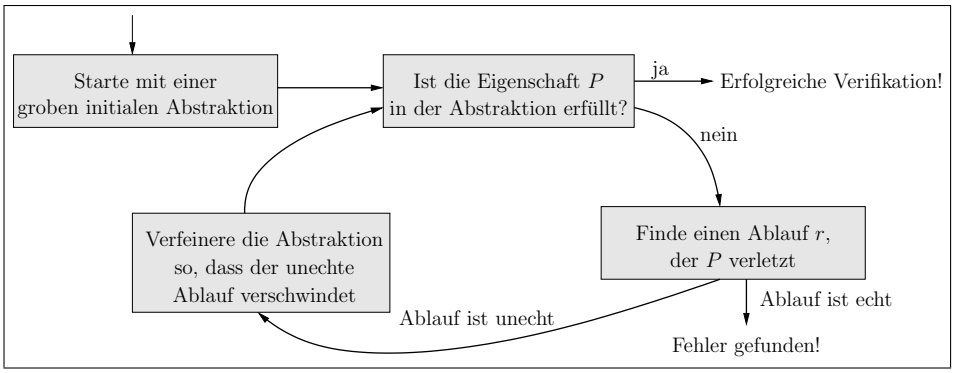


Abbildung 5.7: Abstraktionsverfeinerung basierend auf Gegenbeispielen (Schema)

beispielsweise die jeweiligen schwächsten Vorbedingungen von *false*. Diese Prädikate werden hinzugefügt, um den Verband der abstrakten Werte zu verfeinern.

Dadurch verschwindet tatsächlich das unechte Gegenbeispiel, was sich wie folgt begründen läßt: angenommen, es gibt eine Folge von Übergängen $\langle S_0, \mathit{true}\rangle \Rightarrow \langle S_1, q_1\rangle \Rightarrow \cdots \Rightarrow \langle S_{n-1}, q_{n-1}\rangle \Rightarrow \langle S_n, q_n\rangle$ im abstraktion Transitionssystem, wobei $q_n \not\equiv \mathit{false}$ und im i-ten Schritt jeweils Block C_i passiert wird. Das heißt $q_{i+1} = \overline{sp(q_i, C_{i+1})}$ (wobei $q_0 = \mathit{true}$).

Dann gilt aufgrund der Berechnung von $q_1, \ldots, q_n$ durch stärkste Nachbedingungen, dass

$$\{\mathit{true}\}\, C_1\, \{q_1\} \quad \{q_1\}\, C_2\, \{q_2\} \quad \ldots \quad \{q_{n-2}\}\, C_{n-1}\, \{q_{n-1}\} \quad \{q_{n-1}\}\, C_n\, \{q_n\}$$

Da q_1 die stärkste Nachbedingung von C_1 (bei Vorbedingung *true*) ist, die in $\mathbf{Abs}(p_1, \ldots, p_n)$ liegt, und p_1 eine solche Nachbedingung ist, folgt $q_1 \models p_1$.

Allgemein gilt: wenn $q_i \models p_i$ gilt und man bereits aus p_i folgern kann, dass nach Ausführung von C_{i+1} das Prädikat p_{i+1} gilt, dann muss man dies aus der stärksten Nachbedingung von q_i auch folgern können.[3] Das heißt, es gilt $sp(q_i, C_{i+1}) \models p_{i+1}$. Da q_{i+1} die stärkste Formel ist, die aus $sp(q_i, C_{i+1})$ folgt und im Verband enthalten ist, und p_{i+1} eine solche Formel ist, muss gelten: $q_{i+1} \models p_{i+1}$.

Diese Argumentation kann man auf den ganzen Ablauf anwenden und erhält $q_i \models p_i$ für alle $i \in \{1, \ldots, n\}$, insbesondere gilt für $i \in \{1, \ldots, n-1\}$, dass q_i das Prädikat p_i positiv enthält. Damit erhält man auch $q_n \models \mathit{false}$. Das kann aber nur dann der Fall sein, wenn $q_n \equiv \mathit{false}$ gilt, was ein Widerspruch zur vorherigen Annahme ist. Daraus kann man schließen, dass das Gegenbeispiel durch Abstraktionsverfeinerung verschwindet.

Beispiel 5.2.10 Wir betrachten folgendes mögliche Gegenbeispiel (siehe auch [HJMM04]):

$$[\mathtt{x:=c}]^1;[\mathtt{c:=c+1}]^2;[\mathtt{y:=c}]^3;[\mathtt{x=m}]^4;[\mathtt{y=m}]^5$$

Genau betrachtet sieht man, dass ein solcher Ablauf nicht möglich sein kann. Dies kann man beispielsweise feststellen, indem man die entsprechenden schwächsten Vorbedingungen für die Nachbedingung *false* ermittelt:

$$\{\mathtt{c+1}\neq\mathtt{m}\vee\mathtt{c}\neq\mathtt{m}\}\ [\mathtt{x:=c}]^1\ \{\mathtt{c+1}\neq\mathtt{m}\vee\mathtt{x}\neq\mathtt{m}\}\ [\mathtt{c:=c+1}]^2\ \{\mathtt{c}\neq\mathtt{m}\vee\mathtt{x}\neq\mathtt{m}\}$$
$$[\mathtt{y:=c}]^3\ \{\mathtt{y}\neq\mathtt{m}\vee\mathtt{x}\neq\mathtt{m}\}\ [\mathtt{x=m}]^4\ \{\mathtt{y}\neq\mathtt{m}\}\ [\mathtt{y=m}]^5\ \{\mathit{false}\}$$

Außerdem kann man leicht sehen, dass $\mathtt{(c+1}\neq\mathtt{m}\vee\mathtt{c}\neq\mathtt{m)} \equiv \mathit{true}$ gilt, d.h., das Gegenbeispiel ist unecht.

Ebenso könnte man die stärksten Nachbedingungen – ausgehend von *true* – ermitteln, was zu folgenden Hoare-Tripeln führt:

$$\{\mathit{true}\}\ [\mathtt{x:=c}]^1\ \{\exists\mathtt{x'}(\mathtt{x=c}) \equiv \mathtt{(x=c)}\}\ [\mathtt{c:=c+1}]^2\ \{\exists\mathtt{c'}(\mathtt{x=c'} \wedge \mathtt{c=c'+1}) \equiv \mathtt{(x=c-1)}\}$$
$$[\mathtt{y:=c}]^3\ \{\mathtt{(x=c-1)} \wedge \mathtt{y=c}\}\ [\mathtt{x=m}]^4\ \{\mathtt{(x=c-1)} \wedge \mathtt{y=c} \wedge \mathtt{x=m}\}\ [\mathtt{y=m}]^5\ \{\mathtt{(x=c-1)} \wedge \mathtt{y=c} \wedge \mathtt{x=m} \wedge \mathtt{y=m}\}$$

[3] Eine verwandte Argumentation beruht darauf, dass aus $q \models q'$ immer $sp(q, C) \models sp(q', C)$ folgt (Monotonie).

Wieder kann man zeigen, dass die letzte Bedingung äquivalent zu *false* ist.

Da die ermittelten Vor- bzw. Nachbedingungen manchmal sehr komplex und unnatürlich sind und sich auch teilweise auf Variablen beziehen können, die im bisherigen bzw. im zukünftigen Teilablauf nicht vorkommen, ist man an Verfahren zu ihrer Vereinfachung interessiert. Eines davon ist die sogenannte Craig-Interpolation [McM03, HJMM04].

Definition 5.2.11 (Craig-Interpolant) *Gegeben seien zwei Formeln p, q mit Variablen aus* **Var** *mit $p \models q$. Eine Formel r heißt Craig-Interpolant von p und q, falls folgendes gilt:*

- $p \models r$, $r \models q$ und
- *r enthält nur Variablen, die sowohl in p als auch in q vorkommen.*

Man kann zeigen, dass beispielsweise in der Aussagenlogik (und auch in anderen Logiken) Craig-Interpolanten existieren und algorithmisch bestimmt werden können. Allerdings kann es unter Umständen mehrere Craig-Interpolanten zu zwei Formeln p, q geben. Dann muss einer davon nach vorher festgelegten Kriterien ausgewählt werden.

Craig-Interpolanten werden nun wie folgt eingesetzt. Sei $C_1; \ldots; C_n$ ein unechtes Gegenbeispiel und seien $s_i = sp(true, C_1; \ldots; C_i)$ die stärksten Nachbedingungen (ausgehend von $s_0 = true$) und $w_i = wp(C_{i+1}; \ldots; C_n, false)$ die schwächsten Vorbedingungen (ausgehend von $w_n = false$), wobei jeweils $i \in \{0, \ldots, n\}$. Man kann relativ leicht zeigen, dass $s_i \models w_i$ für alle $i \in \{0, \ldots, n\}$ gelten muss.

Als neue Prädikate wählt man dann zweckmäßigerweise nicht s_i oder w_i sondern einen Craig-Interpolanten c_i von s_i und w_i. Nun ist allerdings auch noch darauf zu achten, dass folgende Bedingung gilt, die auch Tracking-Property genannt wird, erfüllt ist (siehe auch [EKS06]):

$$\{true \equiv c_0\}\, C_1\, \{c_1\} \qquad \{c_1\}\, C_2\, \{c_2\} \qquad \ldots \qquad \{c_{n-2}\}\, C_{n-1}\, \{c_{n-1}\} \qquad \{c_{n-1}\}\, C_n\, \{c_n \equiv false\}$$

Das kann beispielsweise damit erreicht werden, dass man immer den schwächsten oder immer den stärksten Craig-Interpolanten wählt. Damit wird durch Hinzufügen der Prädikate $c_1, \ldots, c_{n-1}$ das unechte Gegenbeispiel eliminiert.

Beispiel 5.2.12 Wir betrachten das unechte Gegenbeispiel aus 5.2.10 und bestimmen die dazugehörigen (stärksten) Craig-Interpolanten. Dabei erhält man:

- $c_0 = true$
- $c_1 =$ `(x=c)`
- $c_2 =$ `(x=c-1)`
- $c_3 =$ `(x=y-1)`
- $c_4 =$ `(m=y-1)`
- $c_5 = false$

Dabei ist zu beachten, dass $c_3 =$ `(x≠y)` auch ein Craig-Interpolant wäre, für den aber das Tracking-Property nicht gelten würde.

Zusätzlich könnte man jede erhaltene Formel in ihre atomaren Prädikate zerlegen, d.h., in Formeln die keine Booleschen Operatoren wie $\wedge$, $\vee$ oder $\neg$ enthalten und vorrangig diese atomaren Prädikate zur Abstraktion hinzufügen.

Eine weitere Optimierung ist die folgende: man beobachtet, dass es ausreicht, Prädikate nur an bestimmten Programmstellen zu verfolgen, um ein Gegenbeispiel zu eliminieren. Insbesondere sind die Craig-Interpolanten nur an den Stellen relevant, für die sie ermittelt wurden. Wenn man also zu einer Art von "lokalen" abstrakten Werten übergeht und zu jeder Programmstelle eine eigene Menge

von Abstraktionen (basierend auf den entsprechenden Prädikten) definiert, so erhält man wesentlich kompaktere und damit einfacher zu analysierende Approximationen. Diese Approximationen sind möglicherweise ungenauer als diejenigen, die man durch die Abstraktion über alle Prädikate erhält, erfüllen jedoch die wichtige Eigenschaft, dass das unechte Gegenbeispiel eliminiert wurde.

Zuletzt behandeln wir noch ein ausführlicheres Beispiel:

Beispiel 5.2.13 (Erweiterter Euklidischer Algorithmus) Betrachten Sie folgendes Programm, dass bei Eingabe zweier Zahlen `a`, `b` ihren größten gemeinsamen Teiler `d` berechnet. Außerdem werden zwei Zahlen `k`, `l` berechnet, für die `k*a + l*b = d` gelten soll. Zu zeigen ist, dass die Bedingung in der letzten If-Anweisung immer erfüllt ist, d.h., dass Block 20 niemals erreicht wird.

```
[m:=a]1;[n:=b]2;[km:=1]3;[lm:=0]4;[kn:=0]5;[ln:=1]6;
while [n≠0]7 do
   if [m≥n]8
      then [m:=m-n]9;[km:=km-kn]10;[lm:=lm-ln]11
      else [exch(m,n)]12;[exch(km,kn)]13;[exch(lm,ln)]14
   fi
od;
[d:=m]15;[k:=km]16;[l:=lm]17;
if [k*a+l*b=d]18
   then [skip]19
   else [skip]20
fi
```

Dabei steht $[\texttt{exch(m,n)}]^{\ell}$ als Abkürzung für $[\texttt{h:=m}]^{\ell_1};[\texttt{m:=n}]^{\ell_2};[\texttt{n:=h}]^{\ell_3}$, wobei h eine Hilfsvariable ist.

Wir betrachten zunächst die gröbstmögliche Abstraktion mit der leeren Prädikatmenge. Dann entspricht die abstrakte Übergangsrelation genau dem Flussgraphen des Programms und es gibt folgenden Pfad zu Block 20:

$$[\texttt{m:=a}]^1;[\texttt{n:=b}]^2;[\texttt{km:=1}]^3;[\texttt{lm:=0}]^4;[\texttt{kn:=0}]^5;[\texttt{ln:=1}]^6;[\texttt{n=0}]^7;[\texttt{d:=m}]^{15};[\texttt{k:=km}]^{16};[\texttt{l:=lm}]^{17};$$
$$[\texttt{k*a+l*b≠d}]^{18};[\texttt{skip}]^{20}$$

Wir beginnen mit *false* als Nachbedingung und berechnen jeweils die schwächsten Vorbedingungen: dabei ergibt sich insbesondere vor Block 15 die Bedingung $q_1 = (\texttt{km*a+lm*b=m})$ und vor Block 1 $r_1 = (\texttt{1*a+0*b=a}\vee\texttt{b≠0}) \equiv \mathit{true}$. Das bedeutet, das Gegenbeispiel ist unecht und die entsprechenden Prädikate – unter ihnen q_1 – werden hinzugefügt.

Danach sind aber noch weitere Gegenbeispiele möglich, insbesondere da die Prädikate keinerlei Aussagen über `n`, `kn` und `ln` machen. Ein weiteres Gegenbeispiel ist das folgende, wobei einmal die While-Schleife und der Else-Fall der If-Anweisung betreten werden:

$$[\texttt{m:=a}]^1;[\texttt{n:=b}]^2;[\texttt{km:=1}]^3;[\texttt{lm:=0}]^4;[\texttt{kn:=0}]^5;[\texttt{ln:=1}]^6;[\texttt{n≠0}]^7;[\texttt{m<n}]^8;$$
$$[\texttt{exch(m,n)}]^{12};[\texttt{exch(km,kn)}]^{13};[\texttt{exch(lm,ln)}]^{14};[\texttt{n=0}]^7;[\texttt{d:=m}]^{15};[\texttt{k:=km}]^{16};[\texttt{l:=lm}]^{17};$$
$$[\texttt{k*a+l*b≠d}]^{18};[\texttt{skip}]^{20}$$

Wenn man *false* als Nachbedingung annimmt und Vorbedingungen berechnet, so erhält man vor Block 12 als einen möglichen Craig-Interpolanten $q_2 = (\texttt{kn*a+ln*b=n})$.

Die Konjunktion der beiden Bedingungen q_1, q_2 ist eine Invariante, die nach Block 6, aber auch bei Eintritt in die Blöcke 9 bzw. 12 gilt. Daher müssen sie auch nach Verlassen der While-Schleife gelten, woraus abgeleitet werden kann, dass die Bedingung in Block 18 immer erfüllt sein muss. Dies kann automatisch durch das Programmanalyseverfahren ermittelt werden.

Wir sehen uns noch einmal genauer an, warum

$$q_1 \wedge q_2 = (\texttt{km*a+lm*b=m}) \wedge (\texttt{kn*a+ln*b=n})$$

eine Invariante ist, d.h., unter den Programmoperationen erhalten bleibt und insbesondere bei jedem Eintritt in die While-Schleife bzw. in the If-Then-Else-Anweisung erfüllt ist.

- Nehmen wir an, $q_1 \wedge q_2$ ist bei Eintritt in Block 9 (Then-Zweig) erfüllt. Wir bezeichnen mit `m'`, `n'`, `km'`, `lm'`, `kn'`, `ln'` die Variablenwerte nach Verlassen von Block 11. Es gilt:

  `m' = m-n` `n' = n` `km' = km-kn` `lm' = lm-ln` `kn' = kn` `ln' = ln`

  Und daraus folgt:

  `km'*a+lm'*b` = `(km-kn)*a+(lm-ln)*b = (km*a+lm*b) - (kn*a+ln*b) = m-n = m'`
  `kn'*a+ln'*b` = `kn*a+ln*b = n = n'`

  Das heißt $q_1 \wedge q_2$ ist erfüllt, wenn Block 11 verlassen wird.

- Nun betrachten wir den Else-Zweig, d.h., wir nehmen an, dass $q_1 \wedge q_2$ bei Eintritt in Block 12 erfüllt ist. Dann gilt für die neuen Variablenwerte:

  `m' = n` `n' = m` `km' = kn` `lm' = ln` `kn' = km` `ln' = lm`

  und damit

  `km'*a+lm'*b` = `kn*a+ln*b = n = m'`
  `kn'*a+ln'*b` = `km*a+lm*b = m = n'`

  Und so kann man auch hier folgern, dass $q_1 \wedge q_2$ bei Verlassen von Block 14 erfüllt ist.

Abstraktionsverfeinerung basierend auf Gegenbeispielen (CEGAR) wie wir es hier beschrieben haben, ist zwar derzeit einer der vielversprechendsten Programmanalyse-Ansätze, aber auch diese Methode hat ihre Grenzen, die derzeit ausgelotet werden. Ein Beispielprogramm, dass sich mit CEGAR in der hier präsentierten Form nicht verifizieren läßt, ist das folgende:

Programm 5.2.14

```
[x:=i]1;[y:=j]2;
while [x≠0]3 do
   [x:=x-1]4;[y:=y-1]5
od;
if [(i=j)∧(y≠0)]6
   then [skip]7
   else [skip]8
fi
```

Intuitiv ist klar, dass Block 7 nicht erreichbar sein kann. Denn wenn nach Beendigung der While-Schleife `i=j` gilt, dann muss wegen `x=0` und der Tatsache, dass `x` und `y` anfangs auf `i` und `j` gesetzt und immer zusammen dekrementiert wurden, auch `y=0` gelten.

Die durch die Abstraktionsverfeinerung mit Craig-Interpolation entstehenden atomaren Prädikate sind `i=j`, `x=`m, `y=`m, `i=`m, `j=`m für alle $m \in \mathbb{N}_0$. Diese beruhen auf immer länger werdenden unechten Gegenbeispielen. Mit diesem Satz von Prädikaten kann die Nicht-Erreichbarkeit von Block 7 jedoch nicht gezeigt werden.

Die benötigte Invariante wäre (`i=j` $\rightarrow$ `x=y`), wofür das Prädikat `(x=y)` notwendig wäre. Dieses wird jedoch durch die Abstraktionsverfeinerung nicht generiert.

Mögliche Lösungen für dieses Problem sind beispielsweise in [JM06] beschrieben.

Anhang A

WHILE-Programme

A.1 Syntax und Semantik von WHILE-Programmen

Um Beweise über die Korrektheit der vorgestellten Analyse-Verfahren führen zu können, benötigten wir zu jeder verwendeten Sprache und zu jedem verwendeten Kalkül eine klar definierte Syntax und Semantik. Im folgenden wird die Syntax und Semantik von WHILE-Programmen beschrieben (siehe [NNH99]).

Arithmetische Ausdrücke Zur Vereinfachung nehmen wir an, dass alle Variablen nur mit Integer-Werten belegt sind. Eine Funktion $\sigma : \mathbf{Var} \to \mathbb{Z}$ beschreibt die Belegung der Menge **Var** der Programmvariablen. Da eine solche Belegung genau den Zustand eines Programms widerspiegelt, wird die Menge aller solcher Belegungen σ mit **State** bezeichnet.

Ein arithmetischer Ausdruck a ist entweder eine Integer-Konstante $z \in \mathbb{Z}$, oder eine Variable $x \in \mathbf{Var}$ oder er besteht aus zwei arithmetischen Ausdrücken a_1, a_2, die durch einen binären Operator *op* verknüpft werden (a_1 *op* a_2). Als binäre Operatoren stehen alle üblichen Rechenoperationen zur Verfügung (`+`,`-`,`*`,`/`, etc.).

Die Menge aller arithmetischen Ausdrücke wird mit **AExp** bezeichnet. Eine Funktion $\mathcal{A}$: **AExp**$\times$ **State** $\to \mathbb{Z}$ dient zur Auswertung arithmetischer Ausdrücke. Falls beispielsweise $\sigma(\mathtt{x}) = 2$ und $\sigma(\mathtt{y}) = -1$ ist, so gilt $\mathcal{A}(\mathtt{x*3\ -\ y}, \sigma) = 2 \cdot 3 - (-1) = 7$.

Boolesche Ausdrücke Ein Boolescher Ausdruck ist entweder der Form $(a_1 = a_2)$, $(a_1 \neq a_2)$ oder $(a_1 \leq a_2)$, wobei a_1 und a_2 arithmetische Ausdrücke sind, oder der Form $\neg b_1$, $b_1 \vee b_2$ bzw. $b_1 \wedge b_2$, wobei b_1 und b_2 Boolesche Ausdrücke sind. Wir werden u.U. bei Bedarf auch andere Vergleichs- und Verknüpfungsoperatoren verwenden. Auch *true* und *false* sind Boolesche Ausdrücke.

Die Menge aller Booleschen Ausdrücke heißt **BExp** und wie bei den arithmetischen Ausdrücken gibt es eine Auswertungsfunktion $\mathcal{B}$: **BExp** $\times$ **State** $\to \{true, false\}$.

Befehle Wir betrachten folgenden kleinen Befehlssatz. Um die Befehle später von außen referenzieren zu können, was sehr wichtig für die Analyse ist, wird jeder Befehl mit einer Markierung ℓ versehen. Diese Markierungen sollten innerhalb eines Programms alle verschieden sein.

Ein Programm S hat deshalb eine der folgenden Formen:

Zuweisung: $[\mathtt{x:=}a]^\ell$ für $x \in \mathbf{Var}$ und $a \in \mathbf{AExp}$.

Skip-Anweisung: $[\mathtt{skip}]^\ell$

Bedingte Anweisung: `if` $[b]^\ell$ `then` S_1 `else` S_2 `fi`, für $b \in \mathbf{BExp}$ und S_1, S_2 sind wiederum Programme

Schleife: `while` $[\mathtt{b}]^\ell$ `do` S `od`, für $b \in \mathbf{BExp}$ und S ist wiederum ein Programm

Sequentielle Komposition: $S_1\,;S_2$, wobei S_1, S_2 wiederum Programme sind

Operationelle Semantik Wir verwenden eine sogenannte Small-Step-Semantik, bei der jeder Schritt einzeln simuliert wird und die dem tatsächlichen Vorgehen bei der Ausführung eines Programms näher kommt. Im Gegensatz dazu wird bei einer Big-Step-Semantik die Semantik einer Schleife durch einen Fixpunkt und damit im wesentlichen durch einen einzigen Schritt beschrieben.

Die Regeln der Semantik sind im wesentlichen selbsterklärend und werden in Tabelle A.1 zusammengestellt. Ein Übergang der Form $\langle S, \sigma\rangle \to \langle S', \sigma'\rangle$ beschreibt wie ein Paar $\langle S, \sigma\rangle$ bestehend aus einem Programm S und einem Zustand σ in ein Restprogramm S' mit Zustand σ' übergeht. Falls das Programm S dabei terminiert, so schreiben wir einfach $\langle S, \sigma\rangle \to \sigma'$.

$$\langle [\texttt{x:=}a]^\ell, \sigma\rangle \to \sigma[\texttt{x} \mapsto \mathcal{A}(a,\sigma)] \quad \textit{[ass]} \qquad \langle [\texttt{skip}]^\ell, \sigma\rangle \to \sigma \quad \textit{[skip]}$$

$$\frac{\langle S_1, \sigma\rangle \to \langle S_1', \sigma'\rangle}{\langle S_1\,\texttt{;}\,S_2, \sigma\rangle \to \langle S_1'\,\texttt{;}\,S_2, \sigma'\rangle} \quad \textit{[seq1]}$$

$$\frac{\langle S_1, \sigma\rangle \to \sigma'}{\langle S_1\,\texttt{;}\,S_2, \sigma\rangle \to \langle S_2, \sigma'\rangle} \quad \textit{[seq2]}$$

$$\langle \texttt{if } [b]^\ell \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ fi}, \sigma\rangle \to \langle S_1, \sigma\rangle \quad \text{falls } \mathcal{B}(b,\sigma) = \textit{true} \quad \textit{[if1]}$$

$$\langle \texttt{if } [b]^\ell \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ fi}, \sigma\rangle \to \langle S_2, \sigma\rangle \quad \text{falls } \mathcal{B}(b,\sigma) = \textit{false} \quad \textit{[if2]}$$

$$\langle \texttt{while } [b]^\ell \texttt{ do } S \texttt{ od}, \sigma\rangle \to \langle S\texttt{;while } [b]^\ell \texttt{ do } S \texttt{ od}, \sigma\rangle \quad \text{falls } \mathcal{B}(b,\sigma) = \textit{true} \quad \textit{[wh1]}$$

$$\langle \texttt{while } [b]^\ell \texttt{ do } S \texttt{ od}, \sigma\rangle \to \sigma \quad \text{falls } \mathcal{B}(b,\sigma) = \textit{false} \quad \textit{[wh2]}$$

Tabelle A.1: Operationelle Semantik von WHILE-Programmen

Aufgabe A.1.1 Schreiben Sie ein WHILE-Programm, das die Fakultätsfunktion berechnet. Dabei soll die Eingabe in der Variable `n` und die Ausgabe in der Variable `f` erfolgen. Führen Sie einige Schritte des Programms aus, für den Fall, dass $\sigma(\texttt{n}) = 3$.

Initiale und finale Labels und die Flussrelation Wir definieren das initiale Label $\mathit{init}(S)$ und die finalen Labels $\mathit{final}(S)$ eines Programms S induktiv wie folgt:

$$\begin{aligned}
\mathit{init}([\texttt{x:=}a]^\ell) &= \ell \\
\mathit{init}([\texttt{skip}]^\ell) &= \ell \\
\mathit{init}(\texttt{if } [b]^\ell \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ fi}) &= \ell \\
\mathit{init}(\texttt{while } [\texttt{b}]^\ell \texttt{ do } S \texttt{ od}) &= \ell \\
\mathit{init}(S_1\,\texttt{;}\,S_2) &= \mathit{init}(S_1)
\end{aligned}$$

$$\begin{aligned}
\mathit{final}([\texttt{x:=}a]^\ell) &= \{\ell\} \\
\mathit{final}([\texttt{skip}]^\ell) &= \{\ell\} \\
\mathit{final}(\texttt{if } [b]^\ell \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ fi}) &= \mathit{final}(S_1) \cup \mathit{final}(S_2) \\
\mathit{final}(\texttt{while } [\texttt{b}]^\ell \texttt{ do } S \texttt{ od}) &= \{\ell\} \\
\mathit{final}(S_1\,\texttt{;}\,S_2) &= \mathit{final}(S_2)
\end{aligned}$$

Des weiteren wird die Flussrelation $\mathit{flow}(S)$ eines Programms wie folgt definiert:

$$
\begin{array}{rcl}
\mathit{flow}([\texttt{x:=}a]^\ell) & = & \emptyset \\
\mathit{flow}([\texttt{skip}]^\ell) & = & \emptyset \\
\mathit{flow}(\texttt{if } [b]^\ell \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ fi}) & = & \mathit{flow}(S_1) \cup \mathit{flow}(S_2) \cup \{(\ell, \mathit{init}(S_1)), (\ell, \mathit{init}(S_2))\} \\
\mathit{flow}(\texttt{while } [\texttt{b}]^\ell \texttt{ do } S \texttt{ od}) & = & \mathit{flow}(S) \cup \{(\ell, \mathit{init}(S))\} \cup \{(\ell', \ell) \mid \ell' \in \mathit{final}(S)\} \\
\mathit{flow}(S_1\,\texttt{;}\,S_2) & = & \mathit{flow}(S_1) \cup \mathit{flow}(S_2) \cup \{(\ell, \mathit{init}(S_2)) \mid \ell \in \mathit{final}(S_1))\}
\end{array}
$$

A.2 Eigenschaften von WHILE-Programmen

Die oben definierten Mengen und Relationen hängen mit der operationellen Semantik von WHILE-Programmen folgendermaßen zusammen:

Satz A.2.1

(a) Falls $\langle S, \sigma\rangle \to \sigma'$ *gilt, dann folgt* $\mathit{final}(S) = \{\mathit{init}(S)\}$.

(b) Falls $\langle S, \sigma\rangle \to \langle S', \sigma'\rangle$, *dann gilt*

i) $\mathit{final}(S) \supseteq \mathit{final}(S')$.

ii) $\mathit{flow}(S) \supseteq \mathit{flow}(S')$.

Beweis: Der Beweis funktioniert in allen Fällen sehr ähnlich. Wir zeigen nur, dass aus $\langle S, \sigma\rangle \to \langle S', \sigma'\rangle$ folgt: $\mathit{flow}(S) \supseteq \mathit{flow}(S')$. Dazu führen wir Induktion über die angewandten Ableitungsregeln.

[if1] In diesem Fall gilt $S = \texttt{if } [\texttt{b}]^\ell \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ fi}$, $S' = S_1$ und $\sigma' = \sigma$. Des weiteren gilt:

$$
\begin{array}{rl}
 & \mathit{flow}(\texttt{if } [b]^\ell \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ fi}) \\
= & \mathit{flow}(S_1) \cup \mathit{flow}(S_2) \cup \{(\ell, \mathit{init}(S_1)), (\ell, \mathit{init}(S_2))\} \\
\supseteq & \mathit{flow}(S_1)
\end{array}
$$

[if2] Analog zum Fall *[if1]*.

[seq1] Es gilt $S = S_1; S_2$, $S' = S_1'; S_2$ und $\langle S_1, \sigma\rangle \to \langle S_1', \sigma'\rangle$. Wir erhalten:

$$
\begin{array}{rl}
 & \mathit{flow}(S_1; S_2) \\
= & \mathit{flow}(S_1) \cup \mathit{flow}(S_2) \cup \{(\ell, \mathit{init}(S_2)) \mid \ell \in \mathit{final}(S_1))\} \\
\supseteq & \mathit{flow}(S_1') \cup \mathit{flow}(S_2) \cup \{(\ell, \mathit{init}(S_2)) \mid \ell \in \mathit{final}(S_1))\} \\
\supseteq & \mathit{flow}(S_1') \cup \mathit{flow}(S_2) \cup \{(\ell, \mathit{init}(S_2)) \mid \ell \in \mathit{final}(S_1'))\} \\
= & \mathit{flow}(S_1'; S_2)
\end{array}
$$

Dabei wurden die Induktionsvoraussetzung und *ii)* verwendet.

[seq2] Es gilt $S = S_1; S_2$, $S' = S_2$ und $\langle S_1, \sigma\rangle \to \sigma'$. Wir erhalten:

$$
\begin{array}{rl}
 & \mathit{flow}(S_1; S_2) \\
= & \mathit{flow}(S_1) \cup \mathit{flow}(S_2) \cup \{(\ell, \mathit{init}(S_2)) \mid \ell \in \mathit{final}(S_1))\} \\
\supseteq & \mathit{flow}(S_2)
\end{array}
$$

[wh2] Es gilt $S = \texttt{while}\ [b]^\ell\ \texttt{do}\ S\ \texttt{od}$, $S' = S\texttt{;while}\ [b]^\ell\ \texttt{do}\ S\ \texttt{od}$ und $\sigma = \sigma'$ und wir erhalten:

$$\begin{aligned}
& \mathit{flow}(S\texttt{;while}\ [b]^\ell\ \texttt{do}\ S\ \texttt{od}) \\
= \ & \mathit{flow}(S) \cup \mathit{flow}(S\texttt{;while}\ [b]^\ell\ \texttt{do}\ S\ \texttt{od}) \\
= \ & \mathit{flow}(S) \cup \mathit{flow}(S) \cup \{(\ell, \mathit{init}(S))\} \cup \{(\ell', \ell) \mid \ell \in \mathit{final}(S)\} \\
= \ & \mathit{flow}(S) \cup \{(\ell, \mathit{init}(S))\} \cup \{(\ell', \ell) \mid \ell \in \mathit{final}(S)\} \\
= \ & \mathit{flow}(S\texttt{;while}\ [b]^\ell\ \texttt{do}\ S\ \texttt{od})
\end{aligned}$$

Die anderen drei Ableitungsregeln beinhalten Terminierung und können nicht angewandt worden sein. □

Anhang B

Ordnungen und Fixpunkte

Eines der wichtigsten mathematische Werkzeuge zur Programmanalyse sind partielle Ordnungen und die dazugehörige Fixpunkt-Theorie. Die folgende kurze Einführung stützt sich in Teilen auf [NNH99].

B.1 Grundlegende Definitionen

Wir beginnen mit den Grundlagen und definieren zunächst den Begriff der Ordnung.

Definition B.1.1 (Ordnung) *Eine* Ordnung *(manchmal auch mit* partieller Ordnung *oder* Halbordnung *bezeichnet) auf einer Menge L ist eine Relation $\sqsubseteq \subseteq L \times L$, für die gilt:*

(a) $\sqsubseteq$ *ist* reflexiv*, d.h., für jedes $l \in L$ gilt $l \sqsubseteq l$.*

(b) $\sqsubseteq$ *ist* transitiv*, d.h., aus $l_1 \sqsubseteq l_2$ und $l_2 \sqsubseteq l_3$ für $l_1, l_2, l_3 \in L$ folgt $l_1 \sqsubseteq l_3$.*

(c) $\sqsubseteq$ *ist* antisymmetrisch*, d.h., aus $l_1 \sqsubseteq l_2$ und $l_2 \sqsubseteq l_1$ für $l_1, l_2 \in L$ folgt $l_1 = l_2$.*

Definition B.1.2 (Obere/untere Schranken und minimale/maximale Elemente) *Sei $\sqsubseteq$ eine Ordnung auf der Menge L und es gelte $Y \subseteq L$. Eine* obere Schranke *von Y ist ein Element $s \in L$ für das gilt: $\forall l \in Y : l \sqsubseteq s$.*

Die kleinste obere Schranke *(oder das Supremum) von Y ist ein Element $s \in L$, das eine obere Schranke von Y ist und für jede andere obere Schranke $s' \in L$ von Y gilt $s \sqsubseteq s'$. Die kleinste obere Schranke von Y wird mit $\bigsqcup Y$ bezeichnet.*

Ein maximales Element *von Y ist ein Element $m \in Y$, so dass für jedes Element $l \in Y$ mit $m \sqsubseteq l$ folgt $m = l$.*

Analog kann man auch eine untere Schranke, die größte untere Schranke oder das Infimum (in Zeichen $\bigsqcap Y$) und minimale Elemente definieren. Für zwei Elemente $l_1, l_2 \in L$ schreibt man auch oft $l_1 \sqcup l_2 = \bigsqcup\{l_1, l_2\}$ und $l_1 \sqcap l_2 = \bigsqcap\{l_1, l_2\}$.

Aufgabe B.1.3

(a) Bestimmen Sie jeweils eine Menge L mit einer partiellen Ordnung $\sqsubseteq$ und eine Teilmenge $Y \subseteq L$, so dass Y folgende Eigenschaften hat.

i) Y besitzt mehrere maximale Elemente.

ii) Y besitzt eine obere Schranke, aber keine kleinste obere Schranke.

iii) Y besitzt eine kleinste obere Schranke, aber kein maximales Element.

(b) Zeigen Sie, dass die kleinste obere Schranke einer Menge Y eindeutig ist, falls sie existiert.

Definition B.1.4 (Vollständiger Verband) *Ein Tupel $(L, \sqsubseteq)$, bestehend aus einer Menge L und einer Ordnung auf L heißt* vollständiger Verband*, wenn jede Teilmenge Y von L eine kleinste obere und eine größte untere Schranke hat. Dies muss insbesondere auch für $Y = \emptyset$ gelten. Man definiert $\bot = \bigsqcap L$ (bottom) und $\top = \bigsqcup L$ (top).*

Dieses Skript beschäftigt sich zumeist mit Potenzmengenverbänden der Form $(\mathcal{P}(X), \subseteq)$, wobei X eine feste Menge ist. Dabei entspricht das Supremum der Vereinigung von Mengen und das Infimum dem Schnitt. Es gibt jedoch noch viele andere Verbände, beispielsweise bilden die reellen Zahlen zwischen 0 und 1 (das Intervall $[0, 1]$) einen Verband mit der $\leq$-Ordnung.

Bemerkung: Vollständige Verbände sind ein Spezialfall von sogenannten *complete partial orders (cpos)*. Bei diesen wird nicht gefordert, dass jede Menge ein Supremum (und Infimum) hat, es reicht, wenn jede gerichtete Menge A ein Supremum hat. Für eine gerichtete Menge A wird gefordert, dass es für alle $a, b \in A$ ein $c \in A$ gibt mit $a \sqsubseteq c$ und $b \sqsubseteq c$.

Aufgabe B.1.5 Beweisen Sie, dass in einem vollständigen Verband immer $\bigsqcup \emptyset = \bigsqcap L$ und $\bigsqcap \emptyset = \bigsqcup L$ gelten.

Wir werden uns des öfteren auf das *Dualitätsprinzip* berufen, das folgendes besagt: wenn $(L, \sqsubseteq)$ ein Verband ist, so ist auch $(L, \sqsupseteq)$ ein Verband. Dabei sind die Infima des einen Verbandes die Suprema des anderen Verbandes und umgekehrt

Definition B.1.6 (Eigenschaften von Ordnungen) *Sei $\sqsubseteq$ eine Ordnung auf der Menge L. Die Ordnung $\sqsubseteq$ heißt* total*, wenn für zwei Element $l_1, l_2 \in L$ immer $l_1 \sqsubseteq l_2$ oder $l_2 \sqsubseteq l_1$ gilt.*

Eine Teilmenge $Y \subseteq L$ heißt Kette*, wenn die Ordnung $\sqsubseteq$ eingeschränkt auf $Y \times Y$ total ist. Eine* aufsteigende Kette *ist eine endliche oder unendliche Folge $l_0, l_1, l_2, l_3, \ldots$ mit $l_i \sqsubseteq l_{i+1}$. Man schreibt auch $(l_n)_{n \in \mathbb{N}_0}$.*

Eine Ordnung erfüllt die Ascending Chain Condition*, wenn es für jede unendliche aufsteigende Kette $(l_n)_n$ einen Index m gibt mit $l_j = l_{j+1}$ für $j \geq m$.*

Analog werden absteigende Ketten und die Descending Chain Condition *definiert. Ordnungen, die die* Descending Chain Condition *erfüllen, werden manchmal auch als* noethersche Ordnungen *bezeichnet.*

Es ist einfach zu sehen, dass jeder endliche Verband die Ascending und die Descending Chain Condition erfüllt.

Definition B.1.7 (Eigenschaften von Funktionen) *Sei $\sqsubseteq_1$ eine Ordnung auf der Menge L_1, $\sqsubseteq_2$ eine Ordnung auf der Menge L_2 und $f \colon L_1 \to L_2$ eine Funktion.*

(a) Die Funktion f heißt monoton*, wenn aus $l_1 \sqsubseteq_1 l_2$ stets $f(l_1) \sqsubseteq_2 f(l_2)$ folgt.*

(b) Falls $(L_1, \sqsubseteq_1)$ und $(L_2, \sqsubseteq_2)$ vollständige Verbände sind, so heißt die Funktion f additiv*, falls für $l, m \in L_1$ stets $f(l \sqcup m) = f(l) \sqcup f(m)$ gilt. Analog heißt eine Funktion f* multiplikativ*, falls für $l, m \in L_1$ stets $f(l \sqcap m) = f(l) \sqcap f(m)$ gilt.*

Lemma B.1.8 *Sei $(L, \sqsubseteq)$ ein Verband, der die Ascending Chain Condition erfüllt. Dann sind für eine Funktion $f \colon L \to L$ die Bedingungen*

(i) f ist additiv

(ii) f ist vollständig additiv, d.h., für alle $Y \subseteq L$, $Y \neq \emptyset$ gilt: $f(\bigsqcup Y) = \bigsqcup \{f(l) \mid l \in Y\}$

äquivalent und in beiden Fällen ist f monoton.

Beweis: Offensichtlich folgt aus Bedingung (ii) Bedingung (i). Außerdem folgt aus Bedingung (i), dass f monoton ist, denn $l_1 \sqsubseteq l_2$ ist gleichbedeutend mit $l_1 \sqcup l_2 = l_2$.

Daher muss nur noch gezeigt werden, dass Bedingung (i) Bedingung (ii) impliziert. Falls Y endlich ist, so gilt $Y = \{y_1, \ldots, y_n\}$ und man erhält:

$$f(\bigsqcup Y) = f(y_1 \sqcup \ldots \sqcup y_n) = f(y_1) \sqcup f(y_2 \sqcup \ldots \sqcup y_n) = f(y_1) \sqcup \ldots \sqcup f(y_n) = \bigsqcup\{f(l) \mid l \in Y\}.$$

Falls allerdings Y unendlich ist, so müssen wir folgendermaßen vorgehen: wir konstruieren eine Kette $(l_n)_n$ wie folgt. Es gilt $l_0 = y_0$ für ein beliebiges Element $y \in Y$. Das Glied l_{n+1} wird wie folgt konstruiert: Falls für alle $y \in Y$ gilt $y \sqsubseteq l_n$, so ist $l_{n+1} = l_n$. Andernfalls wählen wir ein beliebiges $y_{n+1} \in Y$ aus, für das gilt $y_{n+1} \not\sqsubseteq l_n$ und setzen $l_{n+1} = l_n \sqcup y_{n+1}$.

Aufgrund der Ascending Chain Condition stabilisiert sich diese Folge für einen Index m. Das Element l_m ist eine obere Schranke von Y und wegen $l_m = y_0 \sqcup \ldots \sqcup y_m$ und $\{y_0, \ldots, y_m\} \subseteq Y$ ist es auch eine kleinste obere Schranke. Damnach gilt

$$f(\bigsqcup Y) = f(l_m) = f(y_0 \sqcup \ldots \sqcup y_m) = f(y_0) \sqcup \ldots \sqcup f(y_m) \sqsubseteq \bigsqcup\{f(l) \mid l \in Y\}.$$

Außerdem folgt $f(\bigsqcup Y) \sqsupseteq \bigsqcup\{f(l) \mid l \in Y\}$ aus der Monotonie von f. □

Lemma B.1.9 *Sei $\sqsubseteq$ eine Ordnung auf L, die die Ascending Chain Condition erfüllt, und sei $f\colon L \to L$ eine monotone Funktion. Dann gilt für jede aufsteigende Kette $(l_n)_n$: $f(\bigsqcup_{n=0}^{\infty} l_n) = \bigsqcup_{n=0}^{\infty} f(l_n)$, d.h., f ist stetig.*

Beweis: Sei $l_0 \sqsubseteq l_1 \sqsubseteq l_2 \sqsubseteq \ldots$ eine aufsteigende Folge, die aufgrund der Ascending Chain Condition irgendwann stationär werden muss, z.B. für den Index m. Es gilt also $l_m = l_{m+1} = l_{m+2} = \ldots$. Aufgrund der Monotonie von f ist auch $f(l_0), f(l_1), f(l_2), f(l_3), \ldots$ eine aufsteigende Folge, die aufgrund der Ascending Chain Condition stabilisiert, beispielsweise für den Index k. Weil für $j \geq m$ auf jeden Fall gilt $f(l_j) = f(l_{j+1})$, muss gelten $k \leq m$ und außerdem $f(l_k) = f(l_m)$.

Man kann dann folgern:

$$\bigsqcup_{n=0}^{\infty} f(l_n) = f(l_k) = f(l_m) = f(\bigsqcup_{n=0}^{\infty} l_n).$$

□

Außerdem kann man leicht zeigen, dass jede stetige Funktion immer monoton ist (analog zu additiven Funktionen).

B.2 Fixpunktsätze

Jetzt kann der Begriff des Fixpunktes definiert werden.

Definition B.2.1 (Fixpunkte) *Sei $f : L \to L$ eine Funktion auf einem vollständigen Verband $(L, \sqsubseteq)$. Die Menge aller Fixpunkte von f wird definiert als:*

$$Fix(f) = \{l \in L \mid f(l) = l\}.$$

Die Menge aller Präfixpunkte bzw. Postfixpunkte wird folgendermaßen definiert:

$$\begin{aligned} Pre(f) &= \{l \in L \mid f(l) \sqsubseteq l\} \\ Post(f) &= \{l \in L \mid f(l) \sqsupseteq l\} \end{aligned}$$

Des weiteren setzen wir:

$$\begin{aligned} lfp(f) &= \bigsqcap Fix(f) \\ gfp(f) &= \bigsqcup Fix(f) \end{aligned}$$

Wir kommen nun zu dem zentralen Theorem dieses Kapitels, dem Fixpunktsatz von Knaster-Tarski [Tar55]. Er besagt insbesondere, dass für monotone Funktionen die kleinste untere und die größte obere Schranke aller Fixpunkte auch tatsächlich ein Fixpunkt ist. Insbesondere sagt der Satz damit auch aus, dass jede monotone Funktion mindestens einen Fixpunkt besitzt.

Theorem B.2.2 (Knaster-Tarski) *Sei $(L, \sqsubseteq)$ ein vollständiger Verband und $f : L \to L$ eine monotone Funktion. Dann gilt:*

$$\begin{aligned} \mathit{lfp}(f) = \bigsqcap \mathit{Pre}(f) &\quad \in \quad \mathit{Fix}(f) \\ \mathit{gfp}(f) = \bigsqcup \mathit{Post}(f) &\quad \in \quad \mathit{Fix}(f). \end{aligned}$$

Beweis: Wir beweisen diesen Satz für $\mathit{lfp}(f)$, für $\mathit{gfp}(f)$ ist der Beweis analog. Dazu setzen wir $l_0 = \bigsqcap \mathit{Pre}(f)$ und zeigen zunächst $f(l_0) \sqsubseteq l_0$, woraus $l_0 \in \mathit{Pre}(f)$ folgt. Da $l_0 \sqsubseteq l$ für alle $l \in \mathit{Pre}(f)$ gilt und f monoton ist, erhalten wir $f(l_0) \sqsubseteq f(l) \sqsubseteq l$ für alle $l \in \mathit{Pre}(f)$, womit $f(l_0)$ eine untere Schranke für $\mathit{Pre}(f)$ ist. Und weil l_0 die größte untere Schranke von $\mathit{Pre}(f)$ ist, folgt daraus $f(l_0) \sqsubseteq l_0$.

Um $l_0 \sqsubseteq f(l_0)$ zu zeigen, gehen wir wir folgt vor: es gilt $f(f(l_0)) \sqsubseteq f(l_0)$ aufgrund der Monotonie von f. Daraus folgt $f(l_0) \in \mathit{Pre}(f)$ und $l_0 \sqsubseteq f(l_0)$, aufgrund der Definition von l_0.

Insgesamt erhält man dadurch $f(l_0) = l_0$ und $l_0 = \bigsqcap \mathit{Pre}(f)$ ist damit ein Fixpunkt. Nun müssen wir noch zeigen, dass l_0 der kleinste Fixpunkt ist. Das folgt aber daraus, dass $\mathit{Fix}(f) \subseteq \mathit{Pre}(f)$ gilt und l_0 bereits das kleinste Element in $\mathit{Pre}(f)$ ist. □

Für die Existenz des Fixpunkts einer monotonen Funktion reicht es sogar aus, zu fordern, dass man mit einer cpo arbeitet, man benötigt nicht notwendigerweise einen Verband.

Der Satz von Kleene macht hingegen Aussagen darüber, wie ein Fixpunkt bestimmt werden kann, wenn die Funktion gewisse Stetigkeitsbedingungen erfüllt.

Theorem B.2.3 (Kleene) *Sei $(L, \sqsubseteq)$ ein vollständiger Verband und $f : L \to L$ eine monotone Funktion.*

Wenn für jede aufsteigende Kette $(l_n)_n$ gilt: $f(\bigsqcup_{n=0}^{\infty} l_n) = \bigsqcup_{n=0}^{\infty} f(l_n)$, so gilt

$$\mathit{lfp}(f) = \bigsqcup_{n=0}^{\infty} f^n(\bot).$$

Und falls für jede absteigende Kette $(l_n)_n$ gilt: $f(\bigsqcap_{n=0}^{\infty} l_n) = \bigsqcap_{n=0}^{\infty} f(l_n)$, so gilt

$$\mathit{gfp}(f) = \bigsqcap_{n=0}^{\infty} f^n(\top).$$

Beweis: Wir zeigen den Satz wiederum nur für den kleinsten Fixpunkt. Sei $l_0 = \bigsqcup_{n=0}^{\infty} f^n(\bot)$. Wir zeigen zunächst, dass $\bot, f(\bot), f(f(\bot)), \ldots, f^i(\bot), \ldots$ eine aufsteigende Kette ist. Es gilt trivialerweise $\bot \sqsubseteq f(\bot)$ und durch iterierte Anwendung von f auf beiden Seiten folgt mit der Monotonie von f, dass $f^i(\bot) \sqsubseteq f^{i+1}(\bot)$. Es gilt

$$f(l_0) = f(\bigsqcup_{n=0}^{\infty} f^n(\bot)) = \bigsqcup_{n=0}^{\infty} f^{n+1}(\bot) = \bigsqcup_{n=1}^{\infty} f^n(\bot) = \bot \sqcup \bigsqcup_{n=1}^{\infty} f^n(\bot) = \bigsqcup_{n=0}^{\infty} f^n(\bot) = l_0,$$

d.h., l_0 ist tatsächlich ein Fixpunkt.

Wir müssen nun nur noch zeigen, dass l_0 der kleinste Fixpunkt ist. Sei l ein beliebiger Fixpunkt von f, dann gilt: $\bot \sqsubseteq l$. Durch wiederholtes Anwenden von f auf beiden Seiten der Ungleichung erhält man $f^n(\bot) \sqsubseteq l$. Und mit der Definition von l_0 folgt: $l_0 \sqsubseteq l$. □

Daraus ergibt sich bei Ordnungen, die die Ascending Chain Condition erfüllen, folgendes Verfahren zur Berechnung eines (kleinsten) Fixpunkts für eine monotone Funktion f: Zunächst ist f

nach Lemma B.1.9 stetig im Sinne von Theorem B.2.3. Und da $\bot, f(\bot), f^2(\bot), \ldots$ eine aufsteigende Kette ist, gibt es aufgrund der Ascending Chain Condition einen Index m mit $f^m(\bot) = f^{m+1}(\bot)$. Auch für alle $j > m$ muss dann $f^j(\bot) = f^m(\bot)$ gelten. Und nach dem Satz von Kleene gilt dann

$$\mathit{lfp}(f) = \bigsqcup_{n=0}^{\infty} f^n(\bot) = f^m(\bot).$$

D.h., man berechnet sukzessive $f(\bot), f(f(\bot)), \ldots, f^i(\bot), \ldots$ bis die Folge stationär wird und erhält dann den kleinsten Fixpunkt.

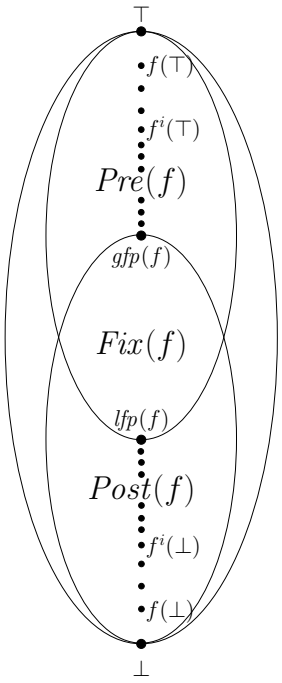


Abbildung B.1: Graphische Visualisierung der Fixpunktsätze von Knaster-Tarski und Kleene

Beispiel B.2.4 Als Beispiel betrachten wir die geordnete Menge $(\mathbb{N}_0, \leq)$ und die Funktion $f\colon \mathbb{N}_0 \to \mathbb{N}_0$ mit $f(n) = n + 1$. Diese Funktion ist monoton, besitzt aber keinen Fixpunkt. Dies liegt daran, dass $(\mathbb{N}_0, \leq)$ kein Verband ist, insbesondere hat die Menge $\mathbb{N}_0$ kein Supremum. Wenn wir aber noch ∞ als Element hinzufügen, so erhält man einen Verband und auch einen Fixpunkt von f.

Außerdem ist jede natürliche Zahl ein Postfixpunkt und das Supremum der Postfixpunkte ist wiederum ∞. Es gibt nur einen Präfixpunkt, nämlich ∞ selbst.

Aufgabe B.2.5

(a) Bestimmen Sie eine Funktion $f\colon L \to L$, die mehrere Fixpunkte, aber keinen kleinsten Fixpunkt, besitzt.

(b) Bestimmen Sie eine monotone Funktion $f\colon L \to L$, die einen kleinsten Fixpunkt l besitzt, so dass aber $\bigsqcup_{n=0}^{\infty} f^n(\bot) \neq l$.

Literaturverzeichnis

[And98] Henrik Reif Andersen. An introduction to binary decision diagrams. Version of October 1997 with minor revisions April 1998, 1998.

[Apt81] Krzysztof R. Apt. Ten years of hoare's logic: A survey – part I. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 3(4):431–483, October 1981.

[BR01] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *International Workshop on SPIN Model Checking '01*, pages 103–122. Springer, 2001. LNCS 2057.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of POPL '77 (Los Angeles, California)*, pages 238–252. ACM, 1977.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proc. of POPL '79 (San Antonio, Texas)*, pages 269–282. ACM Press, 1979.

[CCG+03] Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. In *Proc. of ICSE '03 (25th International Conference on Software Engineering)*, pages 385–395. IEEE Computer Society, 2003.

[CGJ+03] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.

[CGL99] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 1999.

[CGP00] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000.

[Cou96] Patrick Cousot. Abstract interpretation. *ACM Computing Surveys*, 28(2), 1996.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[EKS06] Javier Esparza, Stefan Kiefer, and Stefan Schwoon. Abstraction refinement with craig interpolation and symbolic pushdown systems. In *Proc. of TACAS '06*, pages 489–503. Springer, 2006. LNCS 3920.

[Esp97] J. Esparza. Decidability of model-checking for infinite-state concurrent systems. *Acta Informatica*, 34:85–107, 1997.

[GS97] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In *Proc. of CAV '97*, pages 72–83. Springer, 1997. LNCS 1254.

[HJMM04] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *Proc. of POPL '04*, pages 232–244. ACM, 2004.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Proc. of POPL '02*, pages 58–70. ACM, 2002.

[JM06] Ranjit Jhala and K.L. McMillan. A practical and complete approach to predicate refinement. In *Proc. of TACAS '06*, pages 459–473. Springer, 2006. LNCS 3920.

[JN95] Neil D. Jones and Flemming Nielson. Abstract interpretation: a semantics-based tool for program analysis. In S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science, Vol. 4: Semantic Modelling*, pages 527–636. Oxford University Press, 1995.

[Kle03] Gerwin Klein. *Verified Java Bytecode Verification.* PhD thesis, Institut für Informatik, Technische Universität München, 2003.

[KN01] Gerwin Klein and Tobias Nipkow. Verified lightweight bytecode verification. *Concurrency and Computation: Practice and Experience*, 13:1133–1151, 2001.

[LY99] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification.* Addison-Wesley, 1999. `http://java.sun.com/docs/books/vmspec/`.

[McM03] Kenneth L. McMillan. Interpolation and SAT-based model checking. In *Proc. of CAV '03*, pages 1–13. Springer, 2003. LNCS 2725.

[Nar] Suriya Narayanan. Compiler optimizations in the gnu c compiler. `http://cs.annauniv.edu/ mssnlayam/optimizations/optimizations.html`.

[Nip01] Tobias Nipkow. Verified bytecode verifiers. In *Proc. of FOSSACS '01*, pages 347–363. Springer-Verlag, 2001. LNCS 2030.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer-Verlag, 1999.

[Tar55] Alfred Tarski. A lattice-theoretical theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction.* MIT Press, 1993.