

Abstrakte Interpretation

Ausarbeitung von Bernhard Liebl

(siebtes Fachsemester)

Im Rahmen des Seminars

Nachweis von Sicherheitseigenschaften

für JavaCard durch approximative Programmauswertung

Prof. Tobias Nipkow
Dr. Martin Strecker

Technische Universität München

Department of Computer Science

Inhaltsverzeichnis

| | |
|---------------------------------|-----------|
| 1. EINFÜHRUNG | 3 |
| 1.1. MOTIVATION..... | 3 |
| 1.2. THEMATIK..... | 3 |
| 1.3. EINLEITENDE BEISPIELE..... | 3 |
| 2. FORMALISIERUNG | 5 |
| 2.1. VERBÄNDE..... | 5 |
| 2.2. KONTROLLPUNKTE | 5 |
| 2.3. UMGEBUNGEN | 6 |
| 2.4. DATENFLUßGLEICHUNG | 7 |
| 3. INTERPRETATION | 9 |
| 3.1. INTERVALLVERBÄNDE | 9 |
| 3.2. UMGEBUNGSVERBÄNDE | 10 |
| 3.3. BEISPIEL | 10 |
| 3.4. FIXPUNKTE..... | 12 |
| 3.5. TERMINIERUNG | 12 |
| 4. AUSBLICK..... | 14 |
| 4.1. ANDERE VERBANDSTYPEN | 14 |
| 4.2. ZUSAMMENFASSUNG | 14 |
| 5. LITERATUR..... | 16 |

1. Einführung

1.1. Motivation

In vielen Bereichen moderner Softwareentwicklung besteht der Wunsch, allgemeingültige, von konkreten Eingabewerten entkoppelte Aussagen über Programme zu zeigen, oder aber, besser noch, zu beweisen. Dies ist beispielsweise wichtig für das Finden selten auftretender Fehler (abstraktes Debugging), für den Nachweis von Sicherheitseigenschaften, oder auch für die Erzeugung optimierten Assemblercodes in Compilern.

Insbesondere letzterer Punkt beinhaltet eine Reihe konkreter Anwendungen, wie Konstantenpropagierung, Finden lebender Variablen, Finden toten oder unerreichbaren Codes, oder auch Prüfung von Feldgrenzen oder Typkonvertierungen zur Compile-Zeit. Diese Optimierungen wären sowohl aus Gründen der erhöhten Effizienz des erzeugten Codes als auch, wie beispielsweise bei einer Compile-Zeit-Prüfung für Typkonvertierungen oder für arithmetischen Typüberlauf, aus Sicherheitsüberlegungen heraus wünschenswert.

1.2 Thematik

Bei der als abstrakte Interpretation umschriebenen Methodik im Folgenden geht es darum, für ein gegebenes Programm für alle auftretenden Variablen korrekte und in vertretbarem Aufwand möglichst exakte Abschätzungen anzugeben, welche Werte diese Variablen während eines Programmlaufs annehmen können. Hierbei soll es keine Rolle spielen, wie ein konkreter Programmlauf aussieht, die Analyse soll vielmehr über alle möglichen Fallunterscheidungen, Schleifendurchläufe und extern einlaufenden Parametereingaben eine Art Programmbedeutung ermitteln.

1.3. Einleitende Beispiele

Man betrachte folgendes Beispiel:

```
void debuggingExample()
```

```

{
    int T[ 100 ];
    int n;

    n = readInteger();                                (1)

    for( int i = 0; i < n; i++ )
        T[ i - 1 ] = readInteger();                  (2)

    ...
}

```

In oben aufgeführter Funktion wäre es für einen “schlauem” Compiler möglich, zu erkennen, daß an Stelle (2) für $i = 0$ immer eine Feldgrenzenverletzung stattfindet. Außerdem könnte er warnen, daß wenn bei (1) ein n eingelesen wird, das größer als 101 ist, bei (2) ebenfalls während dem Durchlauf der Schleife auf unzulässige Elemente zugegriffen wird.

Ein weiteres Beispiel:

```

void anotherDebuggingExample()
{
    int i;

    i = readInteger();                                (1)
    j = 2 * i + 1;

    assert( ( j % 2 ) != 0 );                          (2)
}

```

Auch hier könnte ein intelligenter Compiler feststellen, daß der assert bei (2) immer wahr ist, denn das in (1) berechnete j ist hier immer ungerade.

2. Formalisierung

2.1. Verbände

Ein vollständiger Verband $L = (D, \subseteq, \perp, \top, \sqcap, \sqcup)$ definiert über den Operator \subseteq eine Ordnung zwischen jeden zwei Elementen a und b . Jede Untermenge $X \subseteq D$ hat eine kleinste obere Schranke. Ebenso hat jede Untermenge $X \subseteq D$ eine größte untere Schranke.

\perp heißt Bottom, \top heißt Top. Dazu existieren ein sogenannter Infimum-Operator \sqcap und ein Supremum-Operator \sqcup . Diese Operatoren entsprechen in dieser Ausarbeitung eigentlich intuitiv immer dem Schnitt und der Vereinigung von Mengen. Für jedes $x \in L$ gilt außerdem $x \sqcup \top = \top$, sowie $x \sqcap \perp = \perp$.

Zwei Verbände L_1 und L_2 können wie folgt zu einem neuen Verband $L_{1 \times 2}$ kombiniert werden. Ist $L_1 = (D_1, \subseteq_1, \perp_1, \top_1, \sqcap_1, \sqcup_1)$ und $L_2 = (D_2, \subseteq_2, \perp_2, \top_2, \sqcap_2, \sqcup_2)$, so ist $L_{1 \times 2} = (D_1 \times D_2, \subseteq, \perp, \top, \sqcap, \sqcup)$. Die neue Ordnung ist definiert über $(l_1, l_2) \subseteq (l_1', l_2') \iff l_1 \subseteq_1 l_1' \wedge l_2 \subseteq_2 l_2'$. Die restlichen Operatoren werden komponentenweise erweitert. So ist $(l_1, l_2) \sqcap (l_1', l_2') = (l_1 \sqcap_1 l_1', l_2 \sqcap_2 l_2')$ und $(l_1, l_2) \sqcup (l_1', l_2') = (l_1 \sqcup_1 l_1', l_2 \sqcup_2 l_2')$. Schließlich gilt $\top = (\top_1, \top_2)$ und $\perp = (\perp_1, \perp_2)$.

Es sollte klar sein, daß diese Kombination nicht nur für zwei, sondern allgemein für n Verbände funktioniert. Beispielsweise gilt für $L_{1 \times 2 \times \dots \times n}$ die Ordnung $(l_1, \dots, l_n) \subseteq (l_1', \dots, l_n')$ $\iff l_1 \subseteq_1 l_1' \wedge \dots \wedge l_n \subseteq_n l_n'$ und das dazugehörige Bottom-Element ist n -komponentig mit $\perp = (\perp_1, \dots, \perp_n)$.

Schließlich läßt sich für eine Funktion, die auf einem Verband arbeitet, so etwas wie Monotonie definieren. Man definiert, eine Funktion $F: L \rightarrow L$ ist genau dann monoton steigend, wenn $x, y \in L: x \subseteq y \implies F(x) \subseteq F(y)$. In Worten: F ist monoton steigend, wenn die Ordnung aller Paare von Elementen aus L auch unter Funktionsanwendung von F auf die Elemente erhalten wird.

2.2. Kontrollpunkte

Grundsätzlich geht man von einer Darstellung des Programms aus, die das Programm als eine Reihe primitiver Anweisungen beschreibt, wobei jede Anweisung eindeutig einem sogenannten Kontrollpunkt zugeordnet werden kann. Die Kontrollpunkte sollen im Folgenden c_1, \dots, c_n heißen.

```

[ c1 ]
a = 5; [ c2 ]
b = 2; [ c3 ]
a = a + b; [ c4 ]

```

2.3. Umgebungen

Nun überlegt man sich, daß ein Programm Variablen a_1, \dots, a_m benutzt. Der Einfachheit halber nehmen wir an, daß alle Variablen denselben Typ haben. Als eine Umgebung bezeichnet man eine Abbildung von diesen Variablen auf Wertemengen v_1, \dots, v_m , also

$$: \{ a_1 \mapsto v_1, \dots, a_n \mapsto v_m \}$$

Wichtig ist hier anzumerken, daß v_1, \dots, v_m Elemente einer beliebigen abstrakten Domain sein können, solange deren Elemente nur die Werte darstellen (oder auch sammeln) kann, welche Variablen im Rahmen eines Programmlaufs annehmen können. Wir verwenden hier zur Demonstration einfache Mengen, das heißt beispielsweise $v_1 = \{ -7, 193, 715 \}$. Später werden wir die gerade vorgestellten Umgebungen verwenden, um sukzessive alle Werte anzuhäufen, die eine Variable während eines Programmablaufs annehmen kann.

Nun definiert man einen Operator $[\cdot]$, der eine Umgebung in eine neue Umgebung überführen kann, wobei er die durch den Punkt symbolisierte Rechenvorschrift anwendet. Dieser Operator muß alle in der obigen Kontrollpunktdarstellung auftretenden primitiven Anweisungen auswerten können. Beispiel:

$$= \{ a \mapsto \{ 5, 6 \}, b \mapsto \{ 10 \} \}$$

$$\text{' } = [a = a + b]$$

Dann ist ' in diesem Falle

$$\text{' } = \{ a \mapsto \{ 15, 16 \}, b \mapsto \{ 10 \} \}$$

da die Berechnung von $a + b$ unter den Werten, die a und b unter annehmen können, 15

oder 16 ergeben kann. Auch Operatoren wie $[a > 0]$ sind denkbar. Ein solcher Operator würde alle Werte von a „durchlassen“, die größer als null sind, und alle anderen löschen. Nun koppelt man an jede der im vorigen Schritt definierten Kontrollpunkte c_1, \dots, c_n eine Umgebung, und erhält so Umgebungen u_1, \dots, u_n . Im Kontrollpunktbeispiel von oben, würde man folgende Werte erhalten. $?$ stehe hier für einen unbekanntem Wert.

$$\begin{aligned}
 u_1 &= \{ a \mapsto ?, b \mapsto ? \} \\
 u_2 &= \{ a \mapsto \{ 5 \}, b \mapsto ? \} \\
 u_3 &= \{ a \mapsto \{ 5 \}, b \mapsto \{ 2 \} \} \\
 u_4 &= \{ a \mapsto \{ 7 \}, b \mapsto \{ 2 \} \}
 \end{aligned}$$

2.4. Datenflußgleichung

Wie beschreibt man nun den eigentlichen Programmfluß, der Sprünge, Schleifen und bedingte Anweisungen enthalten kann? Für ein Programm ohne Sprünge ergibt sich die Berechnung einer bestimmten Umgebung u_i eindeutig aus einer oder mehrerer Umgebungen, die örtlich vor (und damit mit ihrem Index kleiner) als u_i sind. Mit Sprüngen wird die Sache etwas komplizierter. Beispiel:

```

[ c1 ]
a = 2; [ c2 ]
print a; [ c3 ]
a = a - 1; [ c4 ]
if a > 0 goto c3 [ c5 ]
[ c6 ]

```

In diesem Falle hat die Umgebung u_3 eine Abhängigkeit zu u_4 da zum Zeitpunkt c_3 in a noch 2 stehen kann, jedoch auch ein kleinerer Wert, falls die untere Schleife schon ein- oder mehrmals durchlaufen wurde. Als Funktion ausgedrückt heißt dies $u_3 = f(u_2, u_5)$, denn für die Auswertung von u_3 zu einem gewissen Zeitpunkt müssen wir u_2 und u.U. auch u_5 in Betracht ziehen. Wir haben nun eine Gleichung aufgestellt, die den Schleifenverlauf des

obigen Beispielprogramms charakterisiert. Ausblick: Wären die Umgebungen Verbände, und wollte man alle Werte sammeln, die im Laufe eines Programmablaufs in den Variablen „angehäuft“ werden können, würde man für die obige Gleichung etwas wie $s_3 = s_2 \sqcup s_6$ schreiben.

Allgemein kann man jeden Programmlauf durch eine Reihe solcher Funktionen ausdrücken, also

$$\begin{aligned}
 s_1 &= f_1(s_1, \dots, s_n) \\
 s_2 &= f_2(s_1, \dots, s_n) \\
 &\cdot \\
 &\cdot \\
 &\cdot \\
 s_n &= f_n(s_1, \dots, s_n)
 \end{aligned}$$

Oder, kompakter formuliert,

$$\begin{aligned}
 s_p &= s_1, \dots, s_n \\
 F_p: P &\rightarrow P \\
 F_p &= f_1, \dots, f_n \\
 s_p &= F_p(s_p)
 \end{aligned}$$

Die unterste Zeile entspricht der aufgeschlüsselten Darstellung mit den f_1, \dots, f_n von oben.

3. Interpretation

3.1. Intervallverbände

Im letzten Kapitel wurden die Wertemengen v_1, \dots, v_n durch einfache Mengenschreibweise dargestellt, um die grundsätzliche Idee der Umgebungen zu erklären. Tatsächlich wäre es jedoch wünschenswert, eine formale Struktur zu haben, die gewisse Randbedingungen erfüllt. So wäre eine Ordnung innerhalb der Wertemenge für spätere Fixpunktberechnungen mit diesen Werten von großem Vorteil für die Beweisbarkeit der Terminierung. Außerdem ist auch aus Überlegungen der praktischen Implementation heraus die im vorigen Abschnitt gewählte Mengendarstellung eher ungünstig, da eine effiziente Implementierung sich sehr aufwändig gestalten würde.

Aus diesen Gründen rechnen wir im folgenden nicht mehr mit Wertemengen, sondern nur noch mit Intervallen $[a,b]$, wobei a und b Ganzzahlen aus einer endlichen Menge Z von Ganzzahlen sein sollen. Beispielsweise könnte Z für eine praktische Implementation die Menge aller vorzeichenbehafteten 32-Bit Integerwerte sein.

Nun kann man mit Hilfe der früher vorgestellten Verbandsstruktur die Intervalle $I(Z)$ über Z wie folgt definieren. $I(Z)$ sei ein Verband $Z, \sqsubseteq, \top, \perp, \sqcup, \sqcap$. \perp sei das leere Intervall. \top sei das größte Intervall, also $[-, +]$, wobei $-$ und $+$ hier nicht wirklich unendlich heißen sollen, sondern vielmehr Symbole für Zahlen sein sollen, die kleiner respektive größer als jede in Z auftretende Zahl ist; dies stellt kein Problem dar, da Z ja endlich ist. Betrachtet man nun alle möglichen linken Intervallgrenzen $A = Z \setminus \{-\}$, und alle möglichen rechten Intervallgrenzen $B = Z \setminus \{+\}$, so sei die Menge aller in $I(Z)$ darstellbaren Intervalle also $\{ [l,u] \mid l \in A, u \in B, l < u \}$.

Eine Ordnung läßt sich wie folgt definieren. Ein Intervall a sei in der Ordnung \sqsubseteq des Verbands genau dann kleiner als ein Intervall b , wenn a von b enthalten wird, also $[l_1, u_1] \sqsubseteq [l_2, u_2] \iff l_1 \leq l_2 \wedge u_1 \leq u_2$. Schließlich seien Supremum und Infimum wie Vereinigung und Schnitt von Intervallen definiert, also $[l_1, u_1] \sqcup [l_2, u_2] = [\min(l_1, l_2), \max(u_1, u_2)]$ und $[l_1, u_1] \sqcap [l_2, u_2] = [\max(l_1, l_2), \min(u_1, u_2)]$, falls $\max(l_1, l_2) \leq \min(u_1, u_2)$. Für den Fall daß beim Infimum die letzte Bedingung nicht zutrifft, also kein Schnitt existiert weil die Intervalle disjunkt sind, definieren wir als Ergebnis das leere Intervall, also $[l_1, u_1] \sqcap [l_2, u_2] = \perp$, falls $\max(l_1, l_2) > \min(u_1, u_2)$.

Auf diese Weise haben wir nun eine (endliche) Verbandsstruktur, die alle möglichen Werte darstellen kann, die jede beliebige unserer Variablen a_1, \dots, a_m annehmen kann. Zusätzlich haben wir über den Operator \sqcup ein formales Mittel, um Wertemengen, die aus verschiedenen Ablaufpfaden des Programmes stammen, zu einer neuen Wertemenge zusammenzuführen.

3.2. Umgebungsverbände

Eine Umgebung ist, zur Erinnerung, eine Reihe von Abbildungen von den Programmvariablen auf Wertemengen, in unserem Fall auf Intervallverbände. Nun wissen wir, daß Intervallverbände – wie der Name sagt - Verbände sind, und außerdem, daß sich zwei oder mehrere Verbände wieder zu einem neuen Verband kombinieren lassen. Mit anderen Worten, wenn wir eine Umgebung als Tupel von Intervallverbänden schreiben, so ist klar, daß eine Umgebung auch ein Verband ist. Wir ordnen hierzu die Komponenten im Tupel entsprechend der Reihenfolge unserer Variablen a_1, \dots, a_n .

Wenn Umgebungen Verbände sind, so ist insbesondere auch das in der Datenflußgleichung auftretende $p = \langle \dots \rangle$ ein Verband. Dies bedeutet auch, daß wir für p eine gewisse Ordnung \sqsubseteq angeben können.

3.3. Beispiel

Angenommen wir haben nun ein Programm, daß wie in Abschnitt 2.3. mit Kontrollpunkten versehen und in primitive Anweisungen aufgeschlüsselt ist. Außerdem nehmen wir an, daß wir die Datenflußgleichung aufgestellt haben. Betrachten wir das Beispiel aus 2.4.

```
[ c1 ]  
a = 2; [ c2 ]  
print a; [ c3 ]  
a = a - 1; [ c4 ]  
if a > 0 goto c3 [ c5 ]  
[ c6 ]
```

Die Datenflußgleichung hierfür läßt sich mit Verwendung von Intervallverbänden als Wertemengen wie folgt angeben. Da wir nur eine Variable a haben, sind die Tupel der Umgebungen hier einelementig.

$$\begin{aligned}
f_1(x_1, x_2, x_3, x_4, x_5, x_6) &= \\
f_2(x_1, x_2, x_3, x_4, x_5, x_6) &= [2,2] \\
f_3(x_1, x_2, x_3, x_4, x_5, x_6) &= x_2 \sqcup [a > 0] \quad x_5 \\
f_4(x_1, x_2, x_3, x_4, x_5, x_6) &= [a = a - 1] \quad x_3 \\
f_5(x_1, x_2, x_3, x_4, x_5, x_6) &= x_4 \\
f_6(x_1, x_2, x_3, x_4, x_5, x_6) &= [a = 0] \quad x_5
\end{aligned}$$

Zunächst wissen wir gar nichts über die Variablenwerte, wir setzen also alle Umgebungen $x_i = \mathbb{R}$, oder als Tupel $x_i = (x_{i1}, \dots, x_{im})$ oder ausgeschrieben $x_i = a_1 \mapsto x_{i1}, \dots, a_m \mapsto x_{im}$. Nun berechnen wir der Reihe nach alle neuen x_i ' bis x_n ' durch die Rechenvorschrift $x_i' = x_i \sqcup f_i(x_1, \dots, x_n)$. Wir verwenden eine Vereinigung da wir uns alle Wertebereiche merken wollen, die wir bisher berechnet haben und zu diesen neu berechnete Wertebereiche hinzufügen wollen. So erhalten wir für die erste Iteration

$$\begin{aligned}
x_1 &= \\
x_2 &= [2,2] \\
x_3 &= [2,2] \\
x_4 &= [1,1] \\
x_5 &= [1,1] \\
x_6 &=
\end{aligned}$$

Offensichtlich sind dies nicht die endgültigen Werte. Wenden wir dieselbe Rechenvorschrift wie eben noch einmal an, verändert sich das Bild. Bei nochmaliger Anwendung hingegen bleibt unser Ergebnis gleich. Allgemein betrachtet wendet man die Rechenvorschrift so oft an, bis sich keines der x_i im Vergleich zur Iteration davor mehr ändert. Wir erhalten

$$\begin{aligned}
x_1 &= \\
x_2 &= [2,2] \\
x_3 &= [1,2] \\
x_4 &= [0,1]
\end{aligned}$$

$$s_5 = [0,1]$$

$$s_5 = [0,0]$$

Was wir tatsächlich gerade durchgeführt haben war eine Fixpunktiteration über die Datenflußgleichung. Im nächsten Abschnitt werden wir diese Iteration noch einmal genauer untersuchen, um zu zeigen, daß sie tatsächlich immer terminiert.

3.4. Fixpunkte

Im letzten Abschnitt starteten wir mit der Bottom-Belegung für alle Umgebungen und wendeten dann solange die auf der Datenflußgleichung abgeleiteten Rechenvorschriften an, bis keine Veränderung mehr auftrat. Diese Vorgehensweise entspricht dem Finden eines kleinsten Fixpunktes der Datenflußgleichung $s_p = F_p(s_p)$, denn s_1, \dots, s_n ist genau dann ein Fixpunkt von $F_p = f_1, \dots, f_n$, wenn $f_i(s_1, \dots, s_n) = s_i$ für alle $1 \leq i \leq n$. Innerhalb der Rechnung vereinigten wir Wertemengen verschiedener Programmpfade mittels des Supremum-Operators \sqcup .

Man hätte auch den größten Fixpunkt berechnen können, indem man mit einer Top-Belegung für alle Umgebungen beginnt und dann mittels dem Infimum-Operators \sqcap die Wertemenge solange von oben her "ausdünn", bis ein Fixpunkt erreicht ist. Allerdings gibt es hier keinen Anlaß dazu.

3.5 Terminierung

Viel interessanter ist die Frage, ob die Suche nach einem Fixpunkt nach endlich vielen Schritten terminiert. Um diese Frage zu beantworten, nimmt man die Ordnungen der verwendeten Verbandsstrukturen zu Hilfe.

Nach Definition des Supremum-Operators für Intervallverbände gilt für zwei beliebige Intervalle i_a und i_b daß $i_a \sqsubseteq_i (i_a \sqcup_i i_b)$. Komponentenweise Betrachtung liefert damit auch für zwei beliebige Umgebungen s_a und s_b , daß $s_a \sqsubseteq (s_a \sqcup s_b)$ gilt. Schließlich ist jede Umgebung nur ein Tupel von Intervallverbänden, also $s = i_1, \dots, i_m$.

Betrachtet man nun die Rechenvorschrift zur Berechnung der i-ten Umgebung während der Fixpunktiteration $s_i = s_i \sqcup f_i(s_1, \dots, s_n)$, so ist mit obiger Beobachtung offensichtlich klar, daß s_i zwischen zwei aufeinanderfolgenden Schritten der Fixpunktberechnung niemals

kleiner unter \sqsubseteq werden kann. Intuitiv können wir an einzelnen Kontrollpunkten nur immer noch größere oder zumindest gleich große Intervalle anhäufen, sie jedoch nie kleiner werden lassen.

Nun gibt es genau genommen zwei Fälle, die nach einem Schritt der Fixpunktiteration eintreten können. Entweder hat sich keine einzige Umgebung seit der letzten Iteration geändert. Dann sind wir fertig. Oder aber es hat sich zumindest eine Umgebung geändert. Da wir in der Rechenvorschrift $\mathcal{U}_i' = \mathcal{U}_i \sqcup f_i(\mathcal{U}_1, \dots, \mathcal{U}_n)$ die alte Umgebung mit dem Supremum-Operator berücksichtigen, heißt dies zwangsläufig, daß \mathcal{U}_i' echt größer als \mathcal{U}_i ist, also formal $(\mathcal{U}_i \sqsubseteq \mathcal{U}_i') \wedge (\mathcal{U}_i \neq \mathcal{U}_i')$. Da jedoch jede Umgebung \mathcal{U}_i nur ein endlich großer Verband ist (da wir die Grundwertemenge Z nur endlich groß gewählt haben), kann eine solche Vergrößerung nur endlich oft stattfinden. Mit anderen Worten, nach endlich vielen Iterationsschritten muß der Fall eintreten, daß keine der Umgebungen sich mehr ändert.

4. Ausblick

4.1. Andere Verbandstypen

In dieser Ausarbeitung wurden Intervalle zu einer Verbandsstruktur erweitert und zum Rechnen benutzt. Obwohl diese Struktur einige grundlegende Vorteile hat, kann man sich auch andere verbandartige Strukturen vorstellen.

Zum Beispiel könnte man zum Darstellen einer Wertemenge ein Bitfeld benutzen, wobei jeder mögliche Wert durch ein Bit repräsentiert wird. Vorteilhaft ist diese Struktur, da sie beliebig exakt ist. Die Menge $\{-1234, -20, 50\}$ beispielsweise kann von einem Bitfeld so dargestellt werden, daß genau die drei enthaltenen Zahlen repräsentiert werden. In der Intervalldarstellung hingegen würde man $[-1234, 50]$ schreiben, was natürlich eine große Unschärfe einführt, da viele Zahlen enthalten sind, die eigentlich gar nicht gemeint sind. Speziell für Analysen bei denen es auf Einzelwerte ankommt ist deshalb eine Bitfelddarstellung einer Intervalldarstellung vorzuziehen. Der Nachteil von Bitfelddarstellungen ist der mitunter große Speicherbedarf für große Wertemengen. Abhilfe würde hier eine Begrenzung des Wertebereichs auf interessante Werte schaffen.

Schließlich könnte man sich vorstellen Mehrfachintervalle zu benutzen. Das hieße beispielsweise eine Darstellung wie $[-10,-8] \quad [7,20]$. Das Problem mit dieser Darstellung ist, daß sie in einer praktischen Implementation für Randfälle beinahe beliebig ineffizient wird. Versucht man zum Beispiel alle geraden Zahlen im Intervall $[-10000,10000]$ darzustellen, gerät die Mehrfachintervalldarstellung zu einer sequentiellen Liste einelementiger Teilintervalle.

4.2. Verbesserungen der Laufzeit

Obwohl die Analyse nach endlich vielen Schritten terminiert, ist es für praktische Anwendungen und komplexere Fälle oftmals wünschenswert, Verfahren zu benutzen, die schneller, nach weniger Schritten also, terminieren, als die angegebene naive Auswertung der Fixpunktgleichung. Eine solche Verbesserung ist beispielsweise die Verwendung von sogenannter chaotischer Iteration. Hierbei berechnet man in jedem Iterationsschritt $F(x_1, \dots, x_n) = (x_1, \dots, x_{i-1}, f_i(x_1, \dots, x_n), x_{i+1}, \dots, x_n)$ für ein gewisses $1 \leq i \leq n$, wobei die Wahl eben dieses i der kritische Qualitätsfaktor ist. Eine andere Klasse von Verfahren, um die

Anzahl der notwendigen Iterationen bis zum Erreichen der Fixpunktlösung zu verkleinern, sind sogenannte Narrowing- und Widening-Verfahren. Hierbei werden die Supremum- und Infimum-Operationen auf den Verbänden so modifiziert, daß nach relativ wenigen Widening-Iterationen eine Überapproximation berechnet ist, die dann innerhalb relativ weniger Narrowing-Iterationen wieder verfeinert werden kann. Die genaue Wahl günstiger Widening- und Narrowing-Operatoren soll hier nicht weiter besprochen werden.

4.3. Zusammenfassung

Zusammenfassend läßt sich sagen, daß die Probleme bei der Verwendung der abstrakten Interpretation vor allem in der Unschärfe der Darstellung der Wertemenge und im Berechnungsaufwand für größere Programme mit vielen Schleifen liegen. Auch erschweren Aliasing-Probleme, die durch Zeiger und Referenzen hervorgerufen werden können, die Arbeit massiv. Ein sinnvoller Einsatz abstrakter Interpretation kann nur einhergehen mit Programmiersprachen, die konzeptionell von vorne herein für die Herausforderungen solcher Analysen vorbereitet sind. Speziell eine Anwendung der Technik auf Sprachen wie C wird aus Sicht des Compilerbauers wohl immer unbefriedigend bleiben.

5. Literatur

Cousot, P. (1998). The Calculational Design of a Generic Abstract Interpreter. LIENS, Département de Mathématiques et Informatique.

Elektronisch unter <http://www.di.ens.fr/~cousot/COUSOTpapers/Marktoberdorf98.shtml>.

Bourdoncle, F. (1993). Assertion-based debugging of imperative programs by abstract interpretation. DIGITAL Paris Research Laboratory.

Elektronisch unter <http://www.exentis.com/Francois.Bourdoncle/esec93.pdf>.

Bourdoncle, F. (1993). Efficient chaotic iteration strategies with widenings. DIGITAL Paris Research Laboratory.

Elektronisch unter <http://www.exalead.com/Francois.Bourdoncle/fmpa93.pdf>.

Ermedahl, A., Sjödin, M. (1996). Interval Analysis of C-Variables using Abstract Interpretation. Department of Computer Systems, Uppsala University.

Elektronisch unter http://www.docs.uu.se/~ebbe/abstr_interp/redovisning.ps.