wAnalyzer - ein Tool für die Datenflussanalyse

http://www.fmi.uni-stuttgart.de/szs/tools/wanalyzer

Andreas Gaiser, Frederik Stahr

6. Dezember 2004

Inhaltsverzeichnis

1 Einleitung 5

2 Grundlagen der Datenflussanalyse 7
2.1 Datenflussanalyse und Frameworks . . . 7
2.1.1 Allgemeines . . . 7
2.1.2 Verbände . . . 8
2.1.3 Frameworks . . . 9
2.2 Der Worklist-Algorithmus . . . 11

3 Die WHILE-Sprache von *wAnalyzer* 15
3.1 Sprachbestandteile . . . 15
3.1.1 Datentypen und Variablen . . . 15
3.1.2 Arithmetische und boolesche Ausdrücke . . . 16
3.1.3 Konstrukte . . . 16
3.1.4 Konstrukte für Prozeduren, Unterprogramme oder Funktionen . . . 17
3.2 Programmaufbau . . . 17
3.2.1 Allgemeine Struktur eines Programmes . . . 17
3.2.2 Kommentare . . . 18
3.2.3 Labels . . . 18
3.3 Abschliessendes Beispiel . . . 18

4 Bedienung von *wAnalyzer* 21
4.1 Textbasiert . . . 21
4.2 whrun . . . 23
4.3 Graphisch . . . 24

5 Implementierungsdetails von *wAnalyzer* 27
5.1 Allgemeiner Programmablauf . . . 27
5.1.1 Verarbeiten einer Quelldatei . . . 27
5.1.2 Flussgraph . . . 31

5.2 Der Worklist-Algorithmus 33
5.3 Frameworks 33
5.4 Graphische Ausgabe 36

6 Einbinden eigener Frameworks 39
6.1 Erstellung einer Analyse Schritt für Schritt 39
6.1.1 Charakterisierung der Analyse 40
6.1.2 Die Schablonen “fw/fw.h” und “fw/fw.c” 41
6.1.3 Aktualisieren der “frame_managing.h” 41
6.1.4 Aktualisieren der Make-Datei 42
6.2 Funktionen und Datentypen für die Frameworkerstellung . . . 42
6.2.1 varset.h 42
6.2.2 fwtools.h 43

A Grammatik von *While* 45

B Implementierte Analysearten 47

Kapitel 1

Einleitung

wAnalyzer ist im Rahmen eines Softwarepraktikums im Sommersemester 2004 an der Universität Stuttgart entstanden. Es sollte hauptsächlich in der Lehre eingesetzt werden, um die Prinzipien der Datenflussanalyse und insbesondere die Arbeitsweise des Worklistalgorithmus anschaulich zu machen. Ursprünglich umfasste die Aufgabenstellung folgende Module:

- einen Parser, der aus Programmen einer äußerst einfachen Programmiersprache (`WHILE`) den Syntaxbaum aufbaut
- eine Implementierung der Frameworks, mit denen der Worklistalgorithmus instanziiert wird
- eine Schnittstelle zum Benutzer (textbasiert)

Es existierte schon eine Implementierung des Worklistalgorithmus in der im Softwarepraktikum benutzten Programmiersprache C. Jedoch stellte es sich als günstig heraus, eine eigene zu schreiben, da die bereits vorhandene Implementierung auch für allgemeinere Fälle der Datenflussanalyse (mit Prozeduren etc.) gedacht war. Ausserdem ließ eine eigene Implementierung größere Freiheiten bei der Wahl der Datenstrukturen für die Frameworks, also der eigentlichen Analyseinformationen zu. So ist es auch möglich, in *wAnalyzer* eigene Frameworks ohne hoffentlich zu grossen Aufwand einzubinden, was später erläutert wird. Nach der ersten textbasierten Version von *wAnalyzer* ergab sich die Möglichkeit, auf relativ einfache Weise eine graphische Oberfläche zur Ausgabe des Flussgraphen und seiner abzuarbeitenden Kanten zu erstellen. So besteht *wAnalyzer* nun aus einer textbasierten Version (`wanalyzer`) und einer graphischen (`gwanalyzer`). Als kleines Gimmick wurde ausserdem ein simpler Interpreter für die `WHILE`-Sprache entwickelt, bei dem für Ein- und Ausgabe der Variablen zusätzliche Befehle eingeführt wurden (`PUT`, `GET`): das Programm `whrun`.

Hinweis: Wenn ein Bug entdeckt wurde, wären wir über eine kurze Mail (an stahrfk@studi.informatik.uni-stuttgart.de oder gaiseras@studi.informatik.uni-stuttgart.de), mit einer möglichst detaillierten Beschreibung des Bugs (Reproduzierbarkeit etc.) sehr dankbar. Wenn eine neue Analyse oder gar Codeverbesserungen erstellt wurden, wäre es ebenfalls schön, von diesen zu erfahren, um sie den "offiziellen" Sourcen von *wAnalyzer* hinzuzufügen und somit allen Benutzern zur Verfügung zu stellen. Nun: Viel Spaß :-).

Kapitel 2

Grundlagen der Datenflussanalyse

2.1 Datenflussanalyse und Frameworks

Dieser Abschnitt ist eine sehr allgemeine Einführung in die für die Datenflussanalyse mit dem Worklist-Algorithmus notwendigen mathematischen Werkzeuge, den Verbänden und speziell den monotonen Frameworks. Für eine Einführung in die Datenflussanalyse allgemein, sowie genauere Definitionen und Erläuterungen sei auf [NNH99] und [Kö03] verwiesen.

2.1.1 Allgemeines

Datenflussanalyse wird meist dazu benutzt, Programme zu optimieren, z.B. um unnötige Anweisungen zu entfernen. Dabei wird, wie der Name schon suggeriert, der Fluss von (vorher festgelegten) Daten durch das Programm verfolgt. Es sind viele unterschiedliche Analysen denkbar, jedoch besitzen sie meist zumindest strukturelle Gemeinsamkeiten, die es gestatten, ein einziges Verfahren, den sogenannten Worklist-Algorithmus, zum Gewinnen der Analyseergebnisse einzusetzen. Diesem wird als Argument die Instanz eines monotonen Framework übergeben, die die jeweiligen Informationen über die gewünschte Analyse und ausserdem das zu verarbeitende Programm in Form seines Flussgraphen beinhaltet. Abb. 2.1 zeigt den Flussgraphen für das *WHILE*-Programm

```
WHILE X > 0 DO
  X := X-1;
  IF (A > 0) THEN
    A := X
```

```
  ELSE
    SKIP
  FI
OD
A := 2*A
```

Der Flussgraph enthält nicht sämtliche Informationen des Programmes: Beispielsweise werden alle Ausführungspfade bzw. Verzweigungen als "gleich wahrscheinlich" angesehen (die Kanten eines Flussgraphens sind ungewichtet), es kann also sein, dass Pfade in der Analyse berücksichtigt werden, die eigentlich überhaupt nie vom Programm beschritten werden. Daher sind die Analysen im Allgemeinen unvollständig, jedoch wäre natürlich eine vollständige Analyse turingmächtiger Programme so oder so nicht möglich (Satz von Rice). Von den Analysen wird ausserdem gefordert, dass sie nur einen "einseitigen" Irrtum zulassen, d.h. nie ein falsches Programm als richtig erkennen, sondern höchstens korrekte Programme als falsch klassifizieren.

2.1.2 Verbände

Bei unterschiedlichen Datenflussanalysen werden auch unterschiedliche Aspekte des Programmes untersucht, je nachdem, was analysiert werden soll. So können beispielsweise die Variablen des Programms (bzw. Variablenmengen) oder auch arithmetische Ausdrücke betrachtet werden. Die Menge aller solcher Objekte (z.B. die Menge aller Variablenmengen) muss einem Strukturbegriff genügen, um eine Datenflussanalyse durchführbar zu machen, nämlich dem des vollständigen Verbandes.

Definition 2.1 *Ein vollständiger Verband ist ein Tupel* $(L, \sqsubseteq)$*, bestehend aus einer Menge* L *und einer darauf definierten Ordnung* L*, mit der Eigenschaft, dass jede Teilmenge* Y *von* L *(auch* $\emptyset$*) eine kleinste obere und eine größte untere Schranke haben.*

Für die Analyse mittels des Worklistalgorithmus ist es notwendig, dass jene Analyseobjekte mit einer geeignet gewählten Ordnung einen vollständigen Verband bilden. Die Supremums- bzw. Infimumsfunktionen werden bei der Analyse für das Vereinigen der Analyseergebnisse zweier Knoten benutzt. Abb. 2.2 zeigt ein einfaches Beispiel: Den Flussgraphen für ein Programm der Art

```
IF BED THEN IF-ZWEIG ELSE THEN-ZWEIG FI; SKIP
```

Der Verband besteht im Beispiel aus der Potenzmenge der Variablen als

Grundmenge und der Ordnung $\subseteq$. Im Laufe der Analysebestimmung wird das Analyseergebnis des `SKIP`-Knotens angepasst, und zwar auf das Supremum der Analysewerte der beiden Eingangsknoten, hier entspricht das Supremum also der Vereinigung $\cup$. Die genaue Anwendung wird im Abschnitt über den Worklistalgorithmus besprochen.

2.1.3 Frameworks

Der Verband allein legt nur die Grundmenge einer Analyse fest; auf ein und derselben Grundmenge können jedoch unterschiedliche Analysen durchgeführt werden. Zur Charakterisierung der einzelnen Analysen dient die Struktur des Frameworks bzw. der Frameworkinstanz.

Definition 2.2 *Ein monotoner Framework ist ein Tupel* (L, F)*, wobei* L *ein vollständiger Verband ist und* F *eine Menge von monotonen Funktionen, die die Identität enthält und unter Funktionskomposition abgeschlossen ist.*

Definition 2.3 *Eine* monotone Frameworkinstanz (L, F) *ist ein Tupel* $(L, F,$ *Lab*,*Fluss*, $E, \iota, f.)$*, dessen Komponenten folgende Bedeutung haben:*

- L bezeichnet die Grundmenge des zum Framework gehörenden Verbandes; die Analyseergebnisse sind Elemente von L.

- F ist die Menge von monotonen Funktionen des Frameworks. Aus ihnen rekrutieren sich die Transferfunktionen der Datenflussanalyse, meist werden daher für F Teilmengen der Funktionen der Form $f : L \to L$ verwendet.

- *Lab* bezeichnet die Menge der Labels eines Programms. Jedem Programmblock wird ein Label zugeordnet.

- *Fluss* $\subseteq$ Lab $\times$ Lab bezeichnet die Flussrelation, also die möglichen Abfolgen der Anweisungen des zu analysierenden Programms. Es stellt den schon erwähnten Flussgraphen dar.

- E bezeichnet die Menge der extremalen Labels, die (je nach Analyse) entweder die Endpunkte eines Programms oder der Anfangspunkt sein können.

- ι ist der Wert, der jedem extremalen Label zugeordnet wird. Auch die Anfangswerte der jeweiligen Analyse können höchst unterschiedlicher Natur sein. Zu Beginn des Worklist-Algorithmus werden alle Elemente

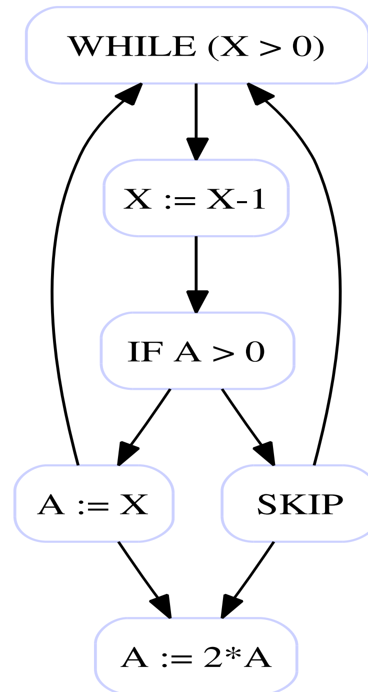

Abbildung 2.1: Flussgraph

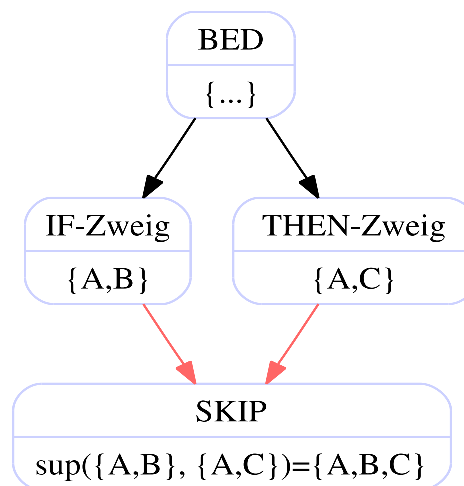

Abbildung 2.2: Beispiel zu Verbänden

aus E mit ι als Analyse initialisiert. Bei der Untersuchung der Lebendigkeit von Variablen ist es z.B. sinnvoll, E jeweils mit den Ausgabevariablen zu initialisieren, da jene am Ende lebendig sein sollen.

- $f.$ bildet jedes Label $l \in \text{Lab}$ auf eine Funktion $f_l \in F$ ab, die die Transferfunktionen darstellen.

2.2 Der Worklist-Algorithmus

Wir wollen ein einfaches WHILE-Programm betrachten:

```
@1: x:=a+b;
@2: while (a > 0) do
@3:   a:=a-1;
@4:   b:=b+a
    od;
@5: x:=b
```

Der zu diesem Programm gehörige Flussgraph ist in Abb. 2.3 dargestellt.

Unter Zuhilfenahme der Labels können wir auch die Flussrelation angeben: $F = \{(1,2),(2,3),(2,5),(3,4),(4,2)\}$. Wollen wir eine Analyse des Programmes durchführen, ist offensichtlich, dass sich beim Passieren eines jeden Knoten des Flussgraphs das Analyseergebnis ändern kann. Wir können also ein Analyseergebnis vor dem Passieren des Knotens von einem nach dem Passieren unterscheiden.
Ersteres wollen wir mit $A_\circ(l)$ und letzteres mit $A_\bullet(l)$ bezeichnen, wobei l die Nummer des Flussgraphknotens bzw. das Label des jeweiligen Blocks ist. Anhand des Flussgraphens können wir nun Gleichungen angeben, in denen die Analyseergebnisse miteinander in Verbindung stehen. Betrachten wir z.B. die Gleichungen einer beliebigen Analyse, die von oben nach unten verläuft:

$A_\circ(1) = \iota$
$A_\bullet(1) = f_1(A_\circ(1))$
$A_\circ(2) = A_\bullet(1) \cup A_\bullet(4)$
$A_\bullet(2) = f_2(A_\circ(2))$
$A_\circ(3) = A_\bullet(2)$
$A_\bullet(3) = f_3(A_\circ(3))$
$A_\circ(4) = A_\bullet(3)$
$A_\bullet(4) = f_4(A_\circ(4))$

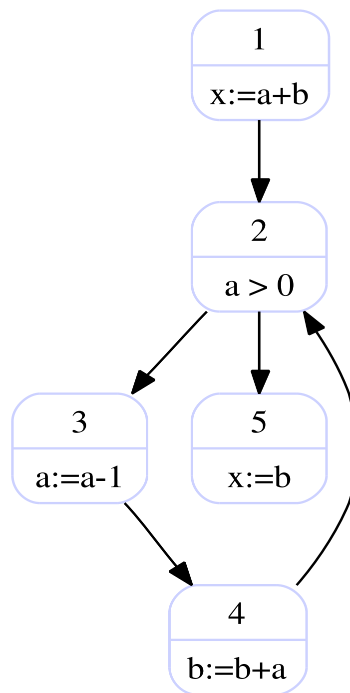

Abbildung 2.3: Flussgraph zum Worklist-Algorithmus

$A_\circ(5) = A_\bullet(4)$
$A_\bullet(5) = f_5(A_\circ(5))$

Die Operation $\cup$ ist im Zusammenhang mit dem jeweiligen Framework zu sehen; sie repräsentiert das Supremum der Verbandsrelation, wie schon weiter oben erwähnt. Wir können erkennen, dass die Gleichungen jeweils voneinander abhängen, also in gewisser Weise "rekursive" Eigenschaften aufweisen. Um sie aufzulösen und so die Analyseergebnisse zu erhalten, kann man das Gleichungssystem auch als Funktion $F : A^{10} \rightarrow A^{10}$ interpretieren (wobei A die Menge der Analyseergebnisse darstellt), für dass

$$F(I_1, I_2, I_3, I_4, I_5, O_1, O_2, O_3, O_4, O_5) =$$
$$(\iota, f_1(I_1), O_1 \cap O_4, f_2(I_2), O_2, f_3(I_3), O_3, f_4(I_4), O_4, f_5(I_5))$$

gilt. Jetzt können wir eine Fixpunktiteration durchführen, die den maximalen (oder je nach Analyse auch den minimalen) Fixpunkt bestimmt, durch Bildung der F^i, solange, bis für ein k gilt: $F^k = F^{k-1}$.

Man kann sich vorstellen, dass eine solche Fixpunktiteration sehr rechenaufwändig ist. Es gibt jedoch eine effiziente Methode, die extremalen Fixpunkte zu bestimmen: den Worklist-Algorithmus. Er benutzt eine Liste, in der zu Anfang alle Kanten des Flussgraphen eingetragen werden, und ein Array, das Analyseergebnisse speichert. Solange die Liste nicht leer ist, wird überprüft, ob die oberste Kante der Liste die vorhandenen Analysewerte verändert; ist dem so, wird das Analyseergebnis angepasst (im Array) und die Kanten, deren Analyseergebnisse sich ebenfalls durch die Korrektur verändert haben könnten, der Liste hinzugefügt. Für tiefergehende Ausführungen, z.B. Korrektheit des Algorithmus oder Bedingungen für ein garantiertes Terminieren einer Fixpunktiteration sei erneut auf [NNH99] und [Kö03] verwiesen.

Im folgenden Pseudocode des Worklistalgorithmus werden Standardprozeduren für Listen verwendet: `cons(a,W)` (hängt `a` an die bestehende Liste W an), `first(`W`)` (liefert das erste Element einer Liste `W`) und `rest(`W`)` (Liefert die Liste W ohne das erste Element). Des weiteren werden die Hilfsvariablen W (die Worklist, enthält Elemente der noch abzuarbeitenden Flussrelation) und `A` (`A[`l`]` ist der bisher berechnete Analysewert für den Eingang zum Block l) benutzt.

Eingabe: Eine (monotone) Frameworkinstanz (L, F,Lab,Fluss, E, ι, f.)
Ausgabe: Die Analyseergebnisse $A_\circ$, $A_\bullet$ für jedes Label

(Initialisierung)

```
W := nil;
forall (l,l') ∈ F do
   W := cons((l',l),W)
od;
forall l ∈ Lab do
   if l ∈ E
      then A[l] := ι
      else A[l] := ⊥
   fi
```

(Iteration)

```
while W ≠ nil do
   (l',l) := first(W);
   W := rest(W);
   if f_l'(A[l']) ⋢ A[l] then
      A[l] := A[l] ⊔ f_l'(A[l']);
      forall l'' with (l,l'') ∈ F do
         W = cons((l,l''),W);
      od
   fi
od
```

(Ausgabe)

```
forall l ∈ Lab do
   A∘(l) := A[l];
   A•(l) := f_l(A[l]);
od
```

Kapitel 3

Die WHILE-Sprache von *wAnalyzer*

Dieses Kapitel stellt eine kurze Zusammenfassung der WHILE-Sprache dar, mit der *wAnalyzer* seine Analysen durchführt. WHILE besitzt nur sehr wenige Programmkonstrukte; Unterprogramme oder Prozeduren fehlen ganz. Es sind lediglich Zuweisungen, einfache Fallunterscheidungen, Schleifen und die leere Anweisung als Befehle möglich. Syntaktisch ähnelt WHILE den Pascal/Ada-Dialekten, jedoch gibt es einige Besonderheiten, auf die hier besonders eingegangen werden soll.

3.1 Sprachbestandteile

3.1.1 Datentypen und Variablen

Der einzige Datentyp, der in *wAnalyzer* verwendet wird, ist `INTEGER`, also ein Teilbereich der ganzen Zahlen. Der exakte Zahlenbereich variiert von System zu System und ist für eine Datenflussanalyse auch nicht unbedingt relevant; er entspricht dem Datentyp `int` der Programmiersprache *C*. Eine Variablendeklaration ist nicht syntaktisch vorgesehen (eine Pseudo-Deklaration bzw. Initialisierung kann z.B. durch eine Anweisung der Form "`VAR:=0`" vorgenommen werden). Alle Variablen sind daraus folgend überall im Programm sichtbar und gültig. Gültige Variablenbezeichner beginnen mit einem Buchstaben, gefolgt von einer (evtl. leeren) Sequenz von Buchstaben, Zahlen oder dem Unterstrich.
Beispiele: "`A`", "`Z_12_Bc`", "`schleife_2`" . Es wird *nicht* zwischen Groß-und Kleinschreibung unterschieden.

3.1.2 Arithmetische und boolesche Ausdrücke

Es stehen die Grundrechenarten (+,-,*,/) zur Verfügung, die beliebig "geschachtelt" werden dürfen. Die Sprache verwendet Infix-Darstellung der Ausdrücke, so dass auch Klammern zur Syntax der Ausdrücke gehören. Als Relationen stehen =,<>,<,>,<= und >= zur Verfügung, außerdem sind einige logische Operationen implementiert: AND (Konjunktion), OR (Disjunktion) und NOT (Negation).
Beispiele für arithmetische und boolesche Ausdrücke:
A+((2*B)/27)-(12*var2), var_a*7 < (2+B)*7,
(A=2) OR (NOT (A<Z+1) AND (B<>2))), TRUE, FALSE.

3.1.3 Konstrukte

Die in *wAnalyzer* verwendete Sprache ist eine erweiterte Form der WHILE-Programme, wie sie aus verschiedenen Büchern der theoretischen Informatik bekannt sind, mit einigen Konstrukten, die die Erstellung eines Programmes vereinfachen. Die einzelnen Bestandteile sind:

- *Zuweisungen*,
- *Fallunterscheidungen* (IF-Konstrukte),
- *Schleifen* (WHILE-Konstrukte),
- *Leere Anweisung* (SKIP-Konstrukte).

Zuweisungen

Syntax: VARNAME := ARITH_EXPR

Hinweise:

- Arithmetische Ausdrücke wurden zuvor besprochen und dargestellt.

Fallunterscheidungen

Syntax: IF BOOL_EXPR THEN PROGRAM_1 ELSE PROGRAM_2 FI

Hinweise:

- BOOL_EXPR kann entweder eine der Konstanten TRUE, FALSE oder ein sonstiger boolescher Ausdruck sein
- Der ELSE-Zweig der Anweisung kann *nicht* weggelassen werden; bei Bedarf muss eine SKIP-Anweisung im ELSE-Zweig eingefügt werden.

Schleifen

Syntax: `WHILE BOOL_EXPR DO PROGRAM_1 OD`

Hinweise:

- `PROGRAM_1` wird ausgewertet, solange `BOOL_EXPR` wahr ist

Leere Anweisung

Syntax: `SKIP`

Hinweise:

- Die leere Anweisung tut buchstäblich nichts; sie ist vor allem dazu da, bei der Datenflussanalyse einen Zwischenschritt zu haben, oder einen nichtbenutzten `THEN`-Zweig zu erlauben.

3.1.4 Konstrukte für Prozeduren, Unterprogramme oder Funktionen

Prozeduren, Unterprogramme oder Funktionen sind keine Bestandteile der WHILE-Sprache und werden daher von *wAnalyzer* nicht unterstützt.

3.2 Programmaufbau

3.2.1 Allgemeine Struktur eines Programmes

Der Aufbau eines WHILE-Programms ist induktiv definiert, und zwar wie folgt:

- Eine Zuweisung oder die leere Anweisung (`SKIP`) ist ein Programm
- Sind S_1 und S_2 Programme, so auch
  - S_1;S_2
  - `IF BOOL_EXPR THEN` S_1 `ELSE` S_2 `FI`
  - `WHILE BOOL_EXPR DO` S_1 `OD`

Man beachte hierbei, dass z.B. "`A:=0; SKIP;`" kein korrektes WHILE-Programm ist, da am Ende eines Programmes nie ein Semikolon steht (das gilt dann per Induktion natürlich auch für Unterzweige in Fallunterscheidungen und Schleifen). Dies ist eine häufige Fehlerquelle, vor allem deshalb, weil das abschließende Semikolon in Sprachen wie z.B. *Pascal* möglich ist, dort wird es dann als eine leere Anweisung interpretiert. Variablendeklarationen sind nicht vorgesehen oder nötig.

3.2.2 Kommentare

Kommentare werden im Stil von *Ada* eingeführt. Zwei Gedankenstriche ("`--`") eröffnen einen Kommentar, der dann bis zum Ende der Zeile dauert. Beispiel hierzu:

```
-- Dies ist ein Kommentar!
```

3.2.3 Labels

Syntax: `@LABEL_NAME:`

Labels dienen nur der Datenflussanalyse; auf eine Ausführung des Programmes haben sie keinerlei Auswirkungen. Insbesondere gibt es in WHILE keine `GOTO`-Anweisungen. Labels können stehen:

- direkt vor einer `IF`-Anweisung
- vor einer Zuweisung
- direkt vor einer `WHILE`-Anweisung
- vor einem `SKIP`-Befehl

Labels sind nicht erlaubt (oder sinnvoll) direkt vor `THEN`, `DO`, `OD` oder `FI`.

3.3 Abschliessendes Beispiel

Abschliessend sei hier ein beispielhaftes WHILE-Programm dargestellt:

```
@ANFANG: y:=a+5;
@L2: x:=2;
@L3: WHILE (y>2) DO
@L4: IF (x<5) THEN
@L5: y:=x
 ELSE
@L6: a:=x-4;
 y:=a+1
 FI;
@L8: y:=y-2
 OD;
@ENDE: y:=7
```

Kapitel 4

Bedienung von *wAnalyzer*

wAnalyzer wird folgendermassen aufgerufen:

```
wanalyzer [-step] <DATEINAME>
```

im Textmodus und

```
gwanalyzer
```

im Graphischen Modus. Dabei ist der Parameter `-step` optional, er veranlasst *wAnalyzer* , beim Abarbeiten des Worklist-Algorithmus bei jedem Schleifendurchlauf auf eine Bestätigung des Benutzers zu warten. Es folgt eine nähere Beschreibung sowohl der textbasierten, als auch der graphisch orientierten Programmvariante samt ihrer Parameter.

4.1 Textbasiert

Die Bedienung der textbasierten Variante soll anhand einer Beispielsitzung erläutert werden. Als Eingabe dient uns folgende Datei:

```
-- lebendig.wh
-- Beispieldatei für lebendige Variablen
@ANFANG:
x := 2;
y := 4;
x := 1;
@VORIF:
if (y > x)
    then z := y
    else z := y*y
```

```
fi;
@ENDE: x := z
```

Diese Datei ist unter "../examples/lebendig.wh" zu finden. Wir wechseln in das `examples`-Verzeichnis und geben folgendes ein:

```
wanalyzer lebendig.wh
```

wAnalyzer wird gestartet, und, soweit das Programm keine Fehler aufweist, wird der Flussgraph erstellt, der Startbildschirm angezeigt und die Auswahl einer Analyseart abgefragt:

```
Datei wird geoeffnet

************************************************
*               wAnalyzer v 0.1                *
************************************************
*                                              *
*                                              *
*    Softwarepraktikum an der Uni Stuttgart    *
************************************************

Flussgraph wird erstellt...
Bitte waehlen Sie eine Analyseart:
[0]:Initialisierung von Variablen
[1]:Lebendige Variablen
[2]:Simple Analysis
[3]:Stark Lebendige Variablen
>
```

Nach Eingabe der gewünschten Ziffer wird der Worklist-Algorithmus mit dem entsprechenden Framework aufgerufen und die Analyse durchführt, ungültige Eingaben beenden *wAnalyzer* .
In unserem Fall möchten wir gerne eine Analyse auf "Lebendige Variablen" durchführen, also geben wir `1` ein. Es erscheint folgendes:

```
Ausgabevariablen auswaehlen aus der Variablenmenge (x y z)
(0 zum Beenden)
```

Bei einigen Analysearten, wie z.B. "Lebendige Variablen" und "Stark lebendige Variablen" wird zusätzlich noch ein für die Analyse spezifischer Wert abgefragt, hier beispielsweise, welche Variablen am Ende des Programms lebendig sein sollen (die "Ausgabe-Variablen"). Wir entscheiden uns für die Variable `x`, also geben wir ein:

```
x 0
```

(die `0` ist das Abschlusszeichen). Nun beginnt *wAnalyzer* die Analyse. Anschließend werden die jeweiligen Analyseergebnisse für die im Programmcode verwendeten Labels ausgegeben, in unserem Falle:

```
Anfagsknoten: 1
Knoten ANFANG [1]
Analyse:
IN: ()
OUT: ()

Knoten VORIF [6]
Analyse:
IN: (x y)
OUT: (y)

Knoten ENDE [7]
Analyse:
IN: (z)
OUT: (x)

Ende
```

Wir können aus der Analyse schliessen, dass die Anweisung mit dem Label `@ANFANG`, also `x := 2`, überflüssig ist: Die veränderte Variable `x` ist am Ausgang von `@ANFANG` nicht lebendig. Zusätzlich können auch noch eigene Frameworks für individuelle Analysearten verwendet werden (s. Kapitel 4). Wird beim Programmaufruf zusätzlich der Paramter "-step" verwendet, so erfolgt die Ausgabe schrittweise für jede Kante des Flussgraphen. Um jeweils einen Schritt weiterzugehen, drücken Sie eine beliebige Taste.

4.2 whrun

Der Interpreter `whrun` wird durch

```
whrun <DATEINAME>
```

aufgerufen. Um eine einfache Ein- und Ausgabe von Variablenwerten zu ermöglichen, wurde die WHILE-Sprache um die Befehle `PUT(ARITH_AUSDRUCK)` und `GET(VARIABLE)` erweitert; einfache Beispiele hierzu sind im `examples`-Verzeichnis zu finden, z.B. die Datei `wurzel.wh`.

4.3 Graphisch

wAnalyzer besitzt ein graphisches Interface, um den Flussgraph und die Analyseverfahren anschaulich zu machen.
Der Aufruf erfolgt mit

```
gwanalyzer
```

Dabei wird die Eingabe eines Dateinamen ignoriert, da dieser erst im darauffolgenden Dialogfenster verlangt bzw. übergeben wird. Im folgenden ist eine kleine Beispielsitzung angegeben. Nach Starten des Programms sollte ein Fenster ähnlich Abb. 4.1 auf dem Bildschirm erscheinen. Klicken Sie nun die Schaltfläche "Laden" bzw. "Load" an, um im darauffolgenden Dateidialog eine zu analysierende WHILE-Quelldatei auszuwählen. Ist das Programm nicht korrekt, wird eine Fehlermeldung ausgegeben; ansonsten baut *wAnalyzer* den Flussgraphen auf und stellt ihn dar (siehe Abb. 4.1 rechts). In unserem Beispiel wurde die Datei "examples/lebendig.wh" ausgewählt.

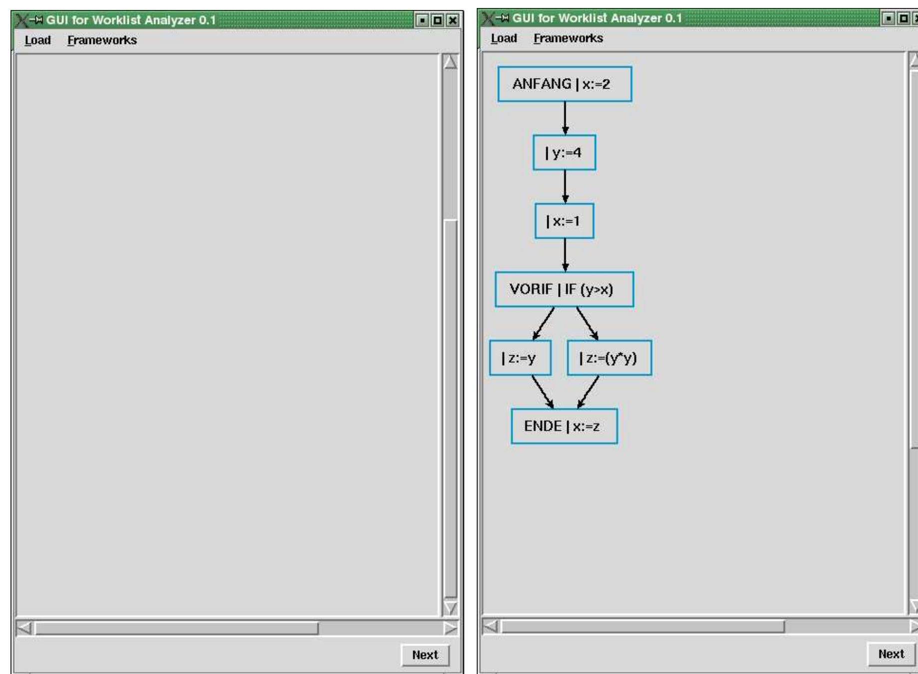

Abbildung 4.1: Startbildschirm (links) und geladene WHILE-Datei (rechts)

Nun kann durch Klicken auf die Schaltfläche "Frameworks" das Menü angezeigt werden, in dem die gewünschte Analyseart ausgewählt werden kann. Bei

manchen Analysen erscheint analog zur Textversion von *wAnalyzer* bei der Auswahl ein Dialogfenster, in dem analysespezifische Daten eingegeben werden müssen; wählen wir z.B. die Analyse "Lebendige Variablen", so erscheint ein Fenster ähnlich dem in Abb. 4.2 .

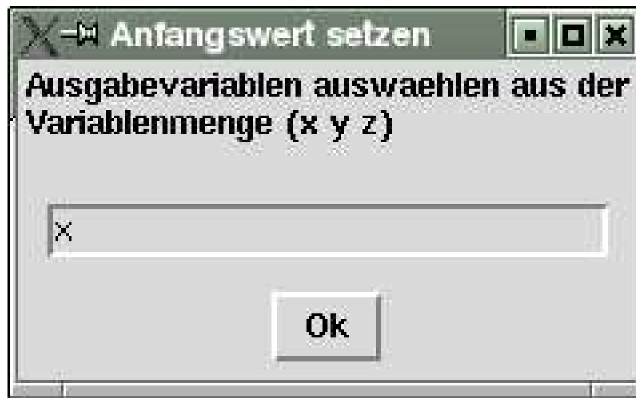

Abbildung 4.2: Dialogbox

Geben Sie die gewünschten Informationen ein (bei Listen von Werten bitte jeden einzelnen Wert durch Leerzeichen getrennt) und klicken Sie auf OK. Der Flussgraph wird mit den analysespezifischen Anfangswerten initialisiert und die Worklist wird aufgebaut. Die oberste Kante in der Worklist wird im Flussgraph rot markiert. Nun können durch Klick auf die Schaltfläche "Next" unten rechts im Fenster die einzelnen Analyseschritte durchgeführt werden, wobei immer die zu bearbeitende Kante rot hervorgehoben wird (siehe Abb. 4.3).

Natürlich können Sie noch weitere Analysen vornehmen und andere Quelldateien laden.

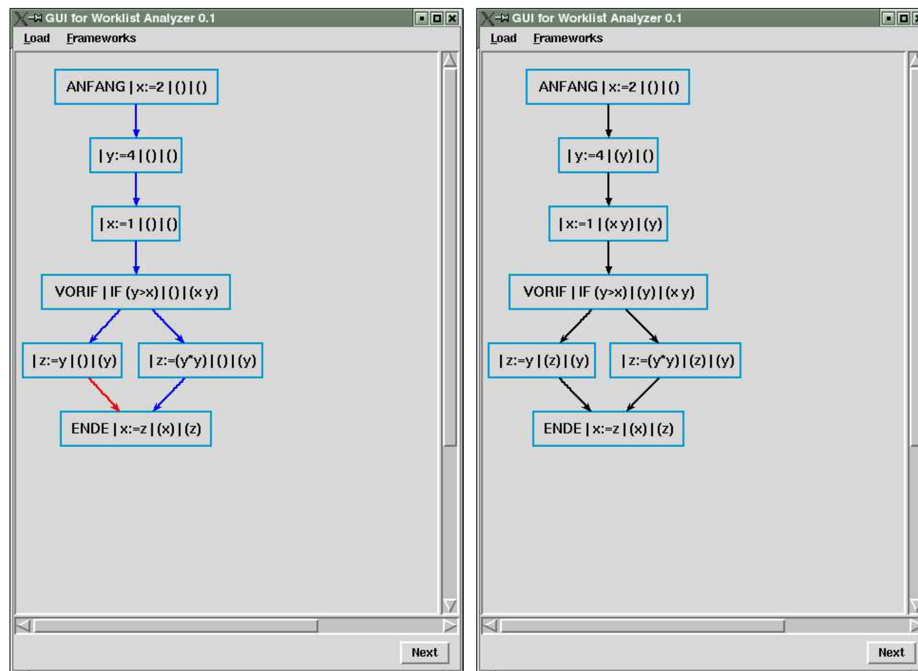

Abbildung 4.3: Analysevorgang: links am Anfang, rechts beendet

Kapitel 5

Implementierungsdetails von *wAnalyzer*

Im folgenden Kapitel sollen die Werkzeuge vorgestellt werden, mit denen *wAnalyzer* entwickelt wurde. Ausserdem wird eine kleine Übersicht über die wichtigen Quelldateien gegeben. Bis auf den Parser und die graphische Oberfläche wurde das gesamte Programm mit der Programmiersprache *C* erstellt; grundlegende Kenntnisse dieser Sprache sind beim Lesen sicher von Vorteil.

5.1 Allgemeiner Programmablauf

5.1.1 Verarbeiten einer Quelldatei

In Abb. 5.1 ist der schematische Ablauf des Auswertens einer Quelldatei mittels *wAnalyzer* angegeben: Zunächst wird eine Quelldatei mittels des Parsermoduls eingelesen. Danach wird, sofern keine Fehler aufgetreten sind, der Parsebaum erstellt. Unter einem Parsebaum verstehen wir im weiteren eine Baumdatenstruktur, in der die Anweisungen sowie die Bestandteile von booleschen und arithmetischen Ausdrücken durch Knoten repräsentiert werden; z.B. besteht ein Parsebaum für ein Programm, das nur aus der Zuweisung `A := B` besteht, aus einem Wurzelknoten, der die Zuweisung an sich repräsentiert, und jeweils zwei Kinderknoten, die die Variablen `A` und `B` repräsentieren. Aus diesem Parsebaum wird bei einer Analyse der Flussgraph generiert, dessen Knoten durch Zeiger mit den korrespondierenden Stellen im Parsebaum verbunden sind, um z.B. arithmetische Ausdrücke aus dem Baum auszuwerten. Ein Flussgraph stellt in gewissem Sinne eine Abstraktion des jeweiligen Programmes dar: Anweisungen und Kontrollstrukturen des

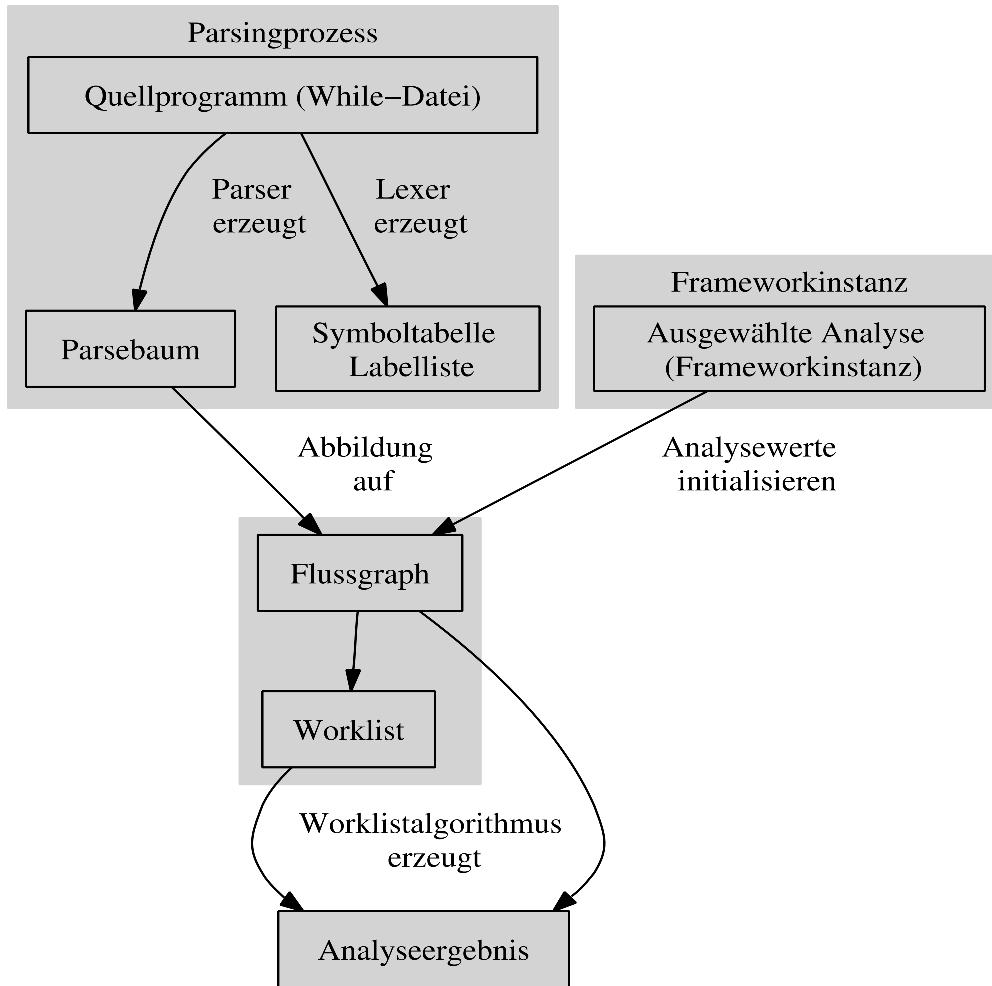

Abbildung 5.1: Programmteile und Ablauf

Programmes bilden die Knoten, die Kanten im Graph korrespondieren mit der schon erwähnten Flussrelation aus **??**.

Lexer und Parser

Die Erstellung des Lexers und Parsers erfolgte mit den Tools *lex* und *yacc*, wer sich näher mit diesen Werkzeugen und ihrer Syntax beschäftigen möchte, sei auf [LMB92] verwiesen. Diese Werkzeuge sind sogenannte Lexer- und Parsergeneratoren. Ein Lexer soll hier eine Prozedur bezeichnen, die die "kleinsten Einheiten" einer Programmiersprache, z.B. Operatoren, Schlüsselwörter etc. erkennt (die sogenannten Tokens) und sie geeignet an eine weitere Prozedur übergibt (geeignet bedeutet hier als Datenstruktur, die die wichtigsten Informationen des Token beinhaltet). Diese weitere Prozedur ist der Parser, in dem die eigentliche Struktur einer Sprache implementiert ist, d.h. syntaktische Regeln, die meist in Form einer (kontextfreien) Grammatik angegeben werden können. *lex* besitzt ein eigenes Eingabeformat für seine Eingabedateien, durch das einzelne Tokens spezifiziert werden können, und zwar in Form von speziellen regulären Ausdrücken. Hinter den Ausdrücken können C-Anweisungen angegeben werden, die dann nach dem Erkennen eines Tokens ausgeführt werden und z.B. eine Instanz der oben erwähnten geeigneten Tokendatenstruktur erstellen. *yacc* parst spezielle kontextfreie Grammatiken, indem es ähnlich wie *lex* eine Grammatikspezifikation einliest und daraus eine C-Prozedur erzeugt, wobei der Programmierer zwischen dem Anwenden der Grammatikregeln Anweisungen einfügen kann. Zum Beispiel befindet sich in unserer Grammatikdefinition von WHILE folgende Zeile:

$$Statement \rightarrow Label\ SimpleStatement \mid SimpleStatement$$

In der Spezifikation des Parsers sieht die dazugehörige Regel folgendermaßen aus:

```
STATEMENT: LABEL SIMPLESTATEMENT { ...C-Code }
| SIMPLESTATEMENT { ...C-Code }
```

Im nächsten Abschnitt wird auf die Funktion des Parsers, nämlich den Parsebaum zu erstellen, genauer eingegangen; die Spezifikation des Lexers ist in "lexer.l", die des Parsers in "parser.y" zu finden. In Lexer und Parser integriert sind Instanzen von Datenstrukturen, die der Verwaltung der Labels und der Variablennamen dienen: nämlich eine `LabelList labellist` und eine `SymbolTable vartable`. Auf diese Variablen wird global von verschiedenen Programmteilen aus zugegriffen. Intern werden Labels und Variablen mit einer `integer`-ID angesprochen, und diese ID wird auch z.B. im

Parsebaum gespeichert (statt des Strings). Um die Bezeichnung des jeweiligen Objekts zu erhalten, stellen LabelList und SymbolTable passende Funktionen bereit, die in "labellist.h" und "symboltable.h" deklariert sind. In einem WHILE-Programm muss nicht jeder Programmblock mit einem Label versehen werden. Das Parser/Lexer-Gespann vergibt aber intern jedem Block eine Label-ID; es muss folglicherweise nicht für jede Label-ID ein Eintrag in `labellist` vorhanden sein. Kleinere technische Probleme zwangen uns, die Datentypdeklaration von `YYSTYPE` in die Datei "parsetree.h" zu packen.

Erstellung des Parsebaums

Eng verzahnt mit dem Parsevorgang geschieht der Aufbau eines Parsebaums vom Typ `ParseTree`, definiert in "parsetree.h". Der Datentyp `ParseTree` soll hier etwas näher erläutert werden:

```
typedef union nodetag_
{
    int value;
    struct ParseTreeList_ *subtrees;
} nodetag;

typedef struct ParseTree_
{
    int type;
    int label;
    nodetag tag;
} *ParseTree;
```

Natürlich stellt die Struktur `ParseTree_` eigentlich nur einen Knoten dar, an dem (eventuell) mehrere Unterbäume hängen können. In den Variablen `type` wird ein Wert für die Art des Knotens abgelegt; die dazugehörigen Konstanten sind in der Datei "symbols.h" gespeichert. `label` speichert die schon erwähnte Label-ID. `nodetag tag` verdient eine etwas genauere Betrachtung: Wenn ein Knoten im Parsebaum ein Blatt darstellt, was z.B. bei Konstanten oder Variablen vorkommt, wird die `value`-Komponente von `tag` benutzt, um z.B. den Wert der Konstanten oder die Variablen-ID zu hinterlegen. Folglich wird bei einem inneren Knoten die Komponente `subtrees` zur Speicherung der Unterbäume verwendet. Für weitere Informationen über implementierte Routinen für `ParseTree` siehe die nachfolgenden Abschnitte.

Um zu demonstrieren, wie der Parser aus dem WHILE-Programm einen Parsebaum aufbaut, betrachten wir noch einmal kurz unser Beispiel einer Grammatikregel aus *yacc*, diesmal mitsamt der angefügten *C*-Anweisungen:

```
STATEMENT: LABEL SIMPLESTATEMENT {nrcommand++;
                                  $$ = $2; $$->label = nrcommand;
                                  labellist =
                                  addLabel(labellist,
                                           nrcommand, $1);}
| SIMPLESTATEMENT {nrcommand++;
                   $$ = $1; $$->label = nrcommand;}
;
```

Hinter jeder Regel kann eine Anweisung bzw. ein Anweisungsblock in *C* angefügt werden; diese Anweisungen werden nach dem Ableiten der einzelnen Terminale oder Nichtterminale (die während ihrer Ableitung natürlich auch wieder Anweisungen "anstoßen") ausgeführt. Natürlich möchte man gerne nach Auswerten bzw. Abarbeiten einer Grammatikregel einen Wert an den Parsingprozess weitergeben; das Startsymbol sollte den Parsebaum für das komplette Programm zurückgeben. In *yacc* werden die Rückgabewerte der einzelnen Nichtterminale durch spezielle Symbole repräsentiert, die alle mit `$` beginnen; in unserem Beispiel repräsentiert `$$` in *C*-Code das Nichtterminal *Statement*, `$1` das erste vorkommende Nichtterminal auf der rechten Seite, also *Label* oder (bei der zweiten Regel) *Statement*, und mit `$2` wird der Rückgabewert des zweiten Nichtterminals (*SimpleStatement*) bezeichnet. Der Datentyp der Nichtterminale ist von *yacc* vordefiniert als `YYSTYPE`, in Fall von *wAnalyzer* eine `union`-Struktur, die entweder einen Parsebaum oder einen Labelbezeichner darstellt. Im Beispiel haben *Statement* und *Simple-Statement* als Wert jeweils einen Parsebaum, während das Nichtterminal *Label* einen String repräsentiert (eben den Namen des Labels). Insofern muss bei Auftreten eines Labels lediglich ein String in `labellist` eingetragen werden, zusammen mit der internen Label-ID des Befehls. Wird ein Label mit angegeben, wird in der Beispielregel noch ein Eintrag in die `Labellist labellist` gemacht.

5.1.2 Flussgraph

Nach dem Erstellen des Parsebaums wird aus diesem der Flussgraph erstellt. Formal gesehen ist für das Ausführen des Worklistalgorithmus lediglich eine Flussrelation nötig, die die einzelnen möglichen Pfade, die eine Programmausführung nehmen kann, beinhaltet. Der in *wAnalyzer* implementierte Flussgraph stellt jedoch mehr als eine Relation auf der Menge der Labels dar, er speichert weitere wichtige Informationen. Die Definition ist in "flownode.h" zu finden:

```
typedef struct FlowNodeList_
```

```
  {
  struct FlowNodeList_ *next;
  struct FlowNode_ *node;
  } *FlowNodeList;

typedef struct FlowNode_
{
    ParseTree expr;
    int label, type;
    Element analysis;
    FlowNodeList entry, exit;
} *FlowNode;

typedef struct FlowGraph_
{
    FlowNode entry;
    FlowNodeList exit;
    FlowNodeList all;
} *FlowGraph;
```

Die Struktur `FlowGraph` enthält zwei Flussgraphknotenlisten: `exit` und `all`. `exit` enthält alle die Knoten, welche Endpunkte des vom Flussgraphen repräsentierten Programms darstellen. `all` enthält alle Knoten des Flussgraphen. `entry` ist der Anfangspunkt des Programmes. In der Struktur `FlowNode` wird ein Zeiger auf den mit dem Knoten korrespondierenden Eintrag im Parsebaum deklariert (`ParseTree expr`); natürlich muss auch das Label und der Typ des jeweiligen Knoten gespeichert werden. Ausserdem wird das Analyseergebnis für den Block abgespeichert: `Element analysis`. Die beiden Flussknotenlisten entsprechen Einträgen im Flussgraph. Lediglich `entry` ist eine Liste, da auch mehrere Knoten Eingänge eines Knoten darstellen können.

Die Erstellung eines Flussgraphen geschieht mittels Prozeduren aus "flownode.h", vornehmlich `FlowGraph generateFlowGraph(ParseTree tree)`. Beispielsweise wird ein Programmblock `IF A THEN P1 ELSE P2` so in einen Flussgraph überführt, dass zunächst die Flussgraphen für die Programmstücke P1 und P2 erstellt werden, dann ein neuer Flussknoten erzeugt wird, der sozusagen für die `IF`-Abfrage steht. Danach werden von diesem IF-Knoten zu den beiden neu erzeugten Flussgraphen für `P1` und `P2` Verbindungen erzeugt. Anfangspunkt des Programmblocks ist nun der `IF`-Knoten, Endpunkte die Vereinigung der Endpunkte von `P1` und `P2`. Dieser rekursive Aufbau wird für alle Programmkonstrukte durchgeführt.

5.2 Der Worklist-Algorithmus

Wie die Implementierung der Frameworks unterscheidet sich auch die Realisierung des Worklistalgorithmus in einigen Punkten von der schon in Abschnitt 2.2 gegebenen Prozedur. Allgemein ist anzumerken, dass der Algorithmisch generisch implementiert ist; er erhält als Eingabe eine Frameworkinstanz (siehe 5.3); diese wiederum enthält Funktionen, die der Algorithmus aufruft, um die Analyseergebnisse zu berechnen. Die Implementierung des Worklistalgorithmus ist in "worklist.h" zu finden, und zwar in zwei Abschnitte unterteilt: Die Prozedur `initWorkList` erstellt die zur jeweiligen Analyse passende Worklist mit den Anfangsanalysewerten, wobei als Worklist der Datentyp `FlowRelationList` verwendet wird, der auch Stackprozeduren wie PUSH und POP anbietet. Die zweite Prozedur `stepWorkList` bekommt den Flussgraph und das aktuelle Framework übergeben und führt einen Schritt des Worklistalgorithmus durch, d.h. er holt eine Kante der Flussrelation aus der Worklist und passt die Analysewerte an. Ist keine Kante mehr in der Worklist, gibt diese Prozedur "1" zurück, ansonsten "0". Zu erwähnen ist, dass eine schon in der Worklist befindliche Kante nicht nochmal in die Worklist aufgenommen wird, jede Kante ist zu jedem Zeitpunkt höchstens einmal in der Worklist.

Wird im Textmodus von *wAnalyzer* die Analyse eines kompletten Programms ausgeführt (nicht im `step`-Modus), so wird `stepWorkList` so lange aufgerufen, bis "0" zurückgegeben wird. Bei der graphischen Version wird bei jedem Klick auf den "Next"-Button `stepWorkList` einmal aufgerufen. Für genauere Informationen zur Implementierung des Algorithmus und der Worklist siehe auch die Dateien "worklist.h" und "flowrel_list.h" bzw. die dazugehörigen *C*-Dateien.

5.3 Frameworks

So wie auch der Flussgraph in *wAnalyzer* nicht einfach eine Relationenliste darstellt, ist auch der der Frameworkinstanz entsprechende Datentyp (zu finden in"framework.h") mehr als das 7-Tupel der formalen Definition. Die jeder Frameworkistanz in *wAnalyzer* zugrundeliegende Datenstruktur besteht hauptsächlich aus mehreren Funktionszeigern. Die Funktionen, die bei der Initialisierung der Frameworkinstanz den Zeigern zugewiesen werden, müssen zuvor ausprogrammiert werden; so ist eine relativ einfache Einbindung weiterer Analysearten möglich. Nachfolgend ist die Deklaration des Datentyps `FrameWork` angegeben (zu finden in "framework.h"):

```
/*
  typedefs for the different
  function pointers
*/
typedef Element (*Iota)();
typedef Element (*Supremum)(Element a, Element b);
typedef Element (*Bottom)();
typedef Element (*Transfer)(FlowNode g, Element a);
typedef int (*InitFunc)();
typedef void (*FinalizeFunc)();
typedef void (*DeRefElement)(Element a);
typedef Element (*RefElement)(Element a);
typedef int (*Eq)(Element a, Element b);
typedef char* (*PrintFunc)(Element a);

/*
  data structure for a general
  framework
*/
typedef struct FrameWork_
{
  /* direction "bit":
     1 for bottom up,
     else top down
  */
  char direction;
  Iota iota;
  /* supremum relation pointer */
  Supremum supremum;
  /* bottom Element */
  Bottom bottom;
  /* the transfer function */
  Transfer transfer;
  /* equality relation */
  Eq eq;
  /* Dereferencing an element */
  DeRefElement deref;
  /* Referencing an element */
  RefElement ref;
  /* Output function */
  PrintFunc printfunc;
```

```
  /* Initialization function */
  InitFunc initfunc;
  /* Finalizer function */
  FinalizeFunc finalizefunc;

} *FrameWork;
```

Um eine höhere Flexibilität in Bezug auf die Analysen zu ermöglichen, wurde eine Art "generische" Datenstruktur erstellt: Die Grundmenge des Verbandes ist gewissermaßen frei wählbar, da alle Funktionen und Prozeduren, die Elemente des Verbandes bearbeiten, nur `void`-Zeiger erwarten (`Element` bezeichnet einen solchen Zeiger), die natürlich auf beliebige Datentypen zeigen können, wie schon erwähnt. Funktionen und Prozeduren des Frameworks, z.B. die Ordnung des Verbandes oder die Transferfunktionen, werden über Funktionszeiger in der Datenstruktur aufgerufen (siehe die `typdefs`); daher kapselt die Datenstruktur alle Charakteristika einer Analyse. Die einzelnen Bestandteile sind (vergleiche dazu auch 2.1.3):

- `char direction` bestimmt die Richtung der Datenflussanalyse und damit auch die Richtung der Pfeile im Flussgraph (Richtung des Flusses); in der formalen Definition ist dieser Wert nicht vorhanden, dort wird die Richtung durch die Flussrelation festgelegt.

- `Iota iota` ist ein Zeiger auf eine 0-stellige Funktion, die die Anfangswerte der Analyse für die extremalen Blöcke bestimmt. Sie entspricht dem ι der formalen Definition.

- `Supremum supremum` ermittelt für je zwei Elemente ein Supremum, das bezügl. der Ordnung des Frameworks größer oder gleich den beiden Elementen ist.

- `Bottom bottom` ist ebenfalls eine 0-stellige Funktion, die den Anfangswert der restlichen Blöcke festlegt.

- `Transfer transfer` bildet das Äquivalent zu den f_l der (formalen) Frameworkinstanzen. Die Funktionenschar wurde ersetzt durch eine einzelne Transferfunktion, die dafür als Argument den jeweiligen Flussgraphknoten (und damit auch dessen bisheriges Analyseergebnis) erhält.

- `Eq eq` stellt eine Gleichheitsrelation für Analyseergebnisse dar, da mit Gleichheit zweier Elemente nicht immer Gleichheit der Speicheradressen gemeint ist.

- `DeRefElement deref` und `RefElement ref` sind (De)referenzierungsprozeduren für Analysewerte. Während des Ausführens des Worklistalgorithmus müssen sehr oft Analyseergebnisse generiert werden, die später nicht mehr benutzt werden. Technisch bedeutet dies, dass Prozeduren bereitgestellt werden müssen, um den Speicherplatz freizugeben, den die Analyseergebnisse beanspruchen. Genau diesem Zweck dienen `DeRefElement deref` und `RefElement ref`. Vor jedem Referenzieren bzw. Deallozieren eines Analysewertes im Worklistalgorithmus werden diese Prozeduren aufgerufen; dadurch lassen sich effiziente Arten der Analyseverwaltung implementieren.

- `PrintFunc printfunc` liefert in einem String die Beschreibung eines Analysewertes, z.B. bei der Grundmenge der Potenzmengen der Variablen können die einzelnen Variablenbezeichner ausgegeben werden.

- `InitFunc initfunc` und `FinalizeFunc finalizefunc` werden jeweils vor dem Durchführen einer Analyse bzw. vor dem Wechseln des aktuellen Frameworks ausgeführt, so dass z.B. globale Variablen initialisiert bzw. am Ende jene wieder freigegeben werden können.

5.4 Graphische Ausgabe

Die graphische Version von *wAnalyzer* benutzt ein Perlskript (mit Tcl/Tk-Anbindung) sowie eine Bibliothek zur Graphenerstellung namens *Graphviz*[1], um die Flussgraphen zu visualisieren. Intern wird von dem Skript zuerst die Datei `wanalyzer` gestartet, und zwar mit dem Argument `-gui`. Dadurch wird *wAnalyzer* in den Graphikmodus versetzt: Es wartet auf Befehle aus der Standardeingabe (die ihm normalerweise natürlich das Skript zusendet), um dann mit einer eigenen Ausgabe über die Standardausgabe zu antworten. Ausgegeben werden hier, im Gegensatz zur textbasierten Version, die kompletten Flussgraphen der Programme, codiert in der Graphendarstellungssprache des Programms *dot* (welches in *Graphviz* enthalten ist). *dot* visualisiert den Graphen und schickt das Bild an das Perlskript zurück, welches wiederum das Bild auf dem Bildschirm ausgibt (via Tcl/Tk-Schnittstelle). Die Befehle, die das Skript (je nach Benutzereingabe) an *wAnalyzer* schickt, werden in `"guictrl.c"` verarbeitet; sie werden nachfolgend aufgeführt.

- `:exit` beendet *wAnalyzer* .

[1]http://www.research.att.com/sw/tools/graphviz

- `:frameworks` veranlasst *wAnalyzer* , eine Liste der installierten und compilierten Analysearten auszugeben, gefolgt von einer Leerzeile.

- `:analysis n` veranlasst *wAnalyzer* die n-te Analyse (mit 0 beginnend) als aktuelle Analyseart auszuwählen; gibt es die Analyse nicht, wird eine Fehlermeldung zurückgegeben. `:analysis` darf nur nach Auswählen einer Datei aufgerufen werden, ansonsten wird ein Fehler zurückgegeben.

- `:filename datei` lässt *wAnalyzer* die entsprechende WHILE-Datei `datei` einlesen; existiert sie nicht, oder enthält sie Fehler, wird eine Fehlermeldung an die Standardausgabe zurückgegeben.

- `:next` kann erst aufgerufen werden, wenn bereits eine Datei und eine Analyseart ausgewählt wurden; der Befehl veranlasst *wAnalyzer* nämlich, einen Schritt des Worklistalgorithmus zu vollziehen, d.h., eine Kante aus der Worklist zu nehmen und die Analyseergebnisse (und die Worklist) entsprechend anzupassen, und selbstverständlich den "neuen" Flussgraphen auszugeben. Ist die Worklist leer, gibt *wAnalyzer* `READY` an die Standardausgabe aus.

Ausser dem `READY`-Signal und den verschiedenen Fehlermeldungen, die alle mit der Zeile `ERROR` beginnen (und als Abschluss ebenfalls eine Leerzeile haben), kann *wAnalyzer* auch eigene Anfragen an das Skript stellen, nämlich durch den Befehl `:prompt`. Hierbei wird ein beschreibender Text (z.B. `Bitte geben Sie die Ausgabevariablen an`) und ein Terminierungszeichen (z.B. `0`), jeweils getrennt durch einen Zeilenumbruch, dem Befehl angefügt. Der Benutzer von *wAnalyzer* bekommt beim Aufruf von `:prompt` (was normalerweise beim Initialisieren einer Analyse geschieht) eine Dialogbox präsentiert, die ihm den Text präsentiert, und zur Eingabe einer Menge von Zeichenketten auffordert. Diese können z.B. Variablenbezeichner sein, oder auch bestimmte andere Initialisierungswerte. Diese Zeichenketten werden dann *wAnalyzer* wieder übergeben, wobei mit einem Zeilenumbruch abgeschlossen wird. Eine Anwendung findet man in der Implementierung der Analyseart "Lebendige Variablen", in der Datei "/fw/fw_livevar.c", unter `int livevar_initfunc()`.

Kapitel 6

Einbinden eigener Frameworks

In *wAnalyzer* sind bereits einige Analysen integriert; jedoch ist es nicht allzu schwierig, weitere Analysen in Form von Frameworkinstanzen dem Programm hinzuzufügen. Diese werden als *C*-Quelldateien hinterlegt und werden somit Teil des Quelltextes des gesamten Programms. *wAnalyzer* stellt einige programmiertechnische Hilfsmittel bereit (z.B. spezielle Datentypen), um die Erstellung solcher Analysen zu vereinfachen; natürlich ist es immer möglich, eigene Datentypen im Framework-Code zu benutzen oder mit jenen auch *wAnalyzer* selbst zu erweitern, z.B. in der Datei "fwtools.h", die später noch erläutert wird. Es empfiehlt sich außerdem, auch das Kapitel 5 zu beachten, da dort wichtige Implementierungsdetails angesprochen werden. Falls die Programmiersprache *C* noch nicht bekannt sein sollte, finden sich in [Wil95] und [Ise01] sehr hilfreiche Ratschläge (leider wurden nicht alle Regeln in *wAnalyzer* eingehalten :-)). *wAnalyzer* wurde so konzipiert, dass das Hinzufügen von Analysen geschehen kann, ohne die Hauptbestandteile des Programms umschreiben zu müssen. Erreicht wird dies durch eine gewisse Generizität des Worklistalgorithmus und der Datenstruktur für die Frameworkinstanzen (siehe 5.2 und 5.3).

6.1 Erstellung einer Analyse Schritt für Schritt

Im folgenden soll die Erstellung eines neuen Frameworks anhand eines Beispiels dargestellt werden, nämlich der schon im Programm integrierten Analyseart *Lebendige Variablen.* Die einzelnen Schritte sollten bei der Erstellung immer eingehalten werden, um Schwierigkeiten beim Build-Prozess des Programms zu vermeiden.

6.1.1 Charakterisierung der Analyse

Am Beginn der Erstellung einer neuen Analyse sollte die genaue Definition der gewünschten Analyseart stehen, was genauer gesagt bedeutet, die einzelnen Bestandteile des dazugehörigen Frameworks zu definieren, und gleichzeitig über ihre Implementierung nachzudenken. Die Frameworkinstanz (bzw. die für uns relevanten Bestandteile) sieht folgendermaßen aus:

Bestandteil	Lebendige Variablen
L	Grundmenge des Frameworks ist die Potenzmenge der Variablenmenge; daher bietet sich der Datentyp `VarSet` an.
$\sqcup$ (Supremum)	$\cup$, also wird die Prozedur `VarSetMerge` benutzt.
$\perp$ (Bottom)	Zu Beginn sind alle Knoten bis auf die extremalen mit $\emptyset$ zu belegen, also mit `NULL`.
E (Extremale Label)	Hier werden die Endknoten mit ι belegt, dies geschieht automatisch durch Setzen von `direction`.
Richtung	Die Analyse geht "bottom-up", daher wird `direction` auf 1 gesetzt.
ι	Der Wert der Endknoten des Flussgraphens hängt davon ab, welche Variablen Ausgabevariablen sind; dies sollte vom Benutzer per `PROMPT` abgefragt werden. Dies gilt sowohl für die graphische wie auch die Textversion (siehe speziell 5.4 für die graph. Ausgabe)

Wichtig ausserdem ist die Transferfunktion, die ja die f_l aus der formalen Definition der Frameworkinstanz repräsentieren. Bei der Analyse "Lebendige Variablen" arbeitet sie wie folgt (abhängig davon, welche Art von Flussgraphknoten übergeben wurde) :

- Wurde ein Knoten übergeben, der eine Bedingung für eine Schleife oder eine `IF`-Anweisung darstellt, wird der aktuelle Analysewert des Knotens mit allen Variablen, die in der Bedingung vorkommen, vereinigt (durch Benutzen der Prozedur `VarSetMerge` und der Hilfsfunktion `fwtools_allVariablesExpr`, die genau alle Variablen aus dem ihr übergebenen Ausdruck ausliest) und diese Vereinigungsmenge zurückgegeben.
- Wurde ein `SKIP` übergeben, so wird der aktuelle Analysewert weitergereicht.

- Bei einer übergebenen Anweisung muss die Variable, deren Wert geändert wurde, aus dem Analyseergebnis entfernt werden (mittels `VarSetremoveElement`) und die Variablen im arithmetischen Ausdruck, der zugewiesen wurde, dem Analyseergebnis hinzugefügt werden (mittels `VarSetMerge`).

6.1.2 Die Schablonen "fw/fw.h" und "fw/fw.c"

Zum Erstellen des Quelltextes für die Frameworkinstanz stellt *wAnalyzer* im Unterverzeichnis "fw/" des Quelltextverzeichnis Schablonendateien bereit ("fw/fw.h" und "fw/fw.c"). In diese wird der Quelltext der Framework-Instanz eingetragen. Um eine gewisse Namenskonvention für Frameworks aufrechtzuerhalten, sollten die `$$$NAME$$$`-Einträge einheitlich mit dem Namen der Analyse überschrieben werden, sowohl in der Header- als auch in der Quelldatei. Die Variable `fw_$$$NAME$$$_name` vom Typ `char[50]` sollte eine möglichst präzise Beschreibung der Analyse beinhalten; dieser String wird nämlich später in der Tabelle der Analysearten angezeigt. Haben wir nun den Code z.B. für unsere Analyse "Lebendige Variablen" erstellt, speichern wir die Datei unter "fw/fw_livevar.h" bzw. "fw/fw_livevar.c" ab.

6.1.3 Aktualisieren der "frame_managing.h"

Nun muss die neue Analyse noch dem Programm selbst bekannt gemacht werden. Analysen werden in der Datei "frame_managing.h" bzw. "frame_managing.c" registriert. Dazu muss zunächst in "frame_managing.h' die Headerdatei der neuen Analyse eingebunden werden, und zwar unterhalb des Kommentars

```
/* include new framework header files here: */
```

Ebenso sollte in dieser Datei der Bezeichnerstring der Analyse als `extern` deklariert werden, nämlich unterhalb von

```
/* identifier strings */
```

Danach muss in "frame_managing.c" ein Eintrag in der Prozedur `init_fwlist()` vorgenommen werden, und zwar unterhalb des Kommentars

```
 /* include FrameWorks here */
```

Der Eintrag sollte folgende Form haben:

```
  registerFrameWork(fwlist,
                    fw_$$$NAME$$$_createFrameWork($$$RICHTUNG$$$),
                    fw_$$$NAME$$$_name);
```

wobei wiederum der richtige Name eingesetzt werden muss und für `$$$RICHTUNG$$$` die Richtung der Analyse (1 für "Bottom up", alles andere für "Top down").

6.1.4 Aktualisieren der Make-Datei

Zu guter Letzt muss noch die Makedatei "makefile" angepasst werden, damit beim nächsten Compilieren auch die neuen Quelldateien übersetzt werden. Dafür sollte ein Eintrag folgender Form erstellt werden:

```
fw_$$$NAME$$$.o : fw/fw_$$$NAME$$$.c fw/fw_$$$NAME$$$.h \
                  symboltable.h varset.h \
              framework.h fwtools.h
              gcc -c fw/fw_$$$NAME$$$.c
```

sowie beim Eintrag `wanalyzer:` jeweils auch ein Zusatz `fw/fw_$$$NAME$$$.o` gemacht werden (sowohl im Bereich der auf Veränderungen zu prüfenden als auch im Bereich des Kompilier-Aufrufs). Nun muss *wAnalyzer* mit dem Befehl `make` neu kompiliert werden, und die neue Analyse müsste einsatzbereit sein.

6.2 Funktionen und Datentypen für die Frameworkerstellung

Um das Erstellen von Frameworks etwas zu erleichtern, sind in *wAnalyzer* einige Datentypen bereits integriert, die für neue Frameworks verwendet werden können, ebenso eine Bibliothek "fwtools.h", in der weitere, speziell für das Verarbeiten von Variablen nützliche Funktionen enthalten sind.

6.2.1 varset.h

Die Bibliothek varset bzw. der in ihr deklarierte Datentyp `VarSet` implementiert eine Variablenmenge (eigentlich eine Liste von Variablen-IDs); für sehr viele Analysen bilden Variablenmengen die Grundmenge des Frameworks. Eingebunden wird die Bibliothek mit `#include "varset.h"`. Initialisiert wird ein neues `VarSet v` durch `v = createVarSet();`, freigegeben wird es durch `VarSetDestroy(v)`. Es muss noch erwähnt werden, dass alle Operationen bis auf das einzelne Hinzufügen eines Elementes nicht-destruktiv sind; wird z.B. eine Vereinigung zweier `VarSet`s durchgeführt, so wird eine komplett neue Instanz angelegt, die mit den ursprünglichen keinerlei Elemente

gemeinsam hat. Darum ist es auch wichtig, bei Verwendung von `VarSet` auf Speicherlecks zu achten bzw. nicht mehr benötigte Instanzen wieder freizugeben. Benötigt man eine Auflistung aller Variablen eines Programms, so ist diese Liste wiederum in "fwtools.h" zu finden: die globale Variable `VarSet var_all` Für eine Dokumentation der einzelnen Funktionen sei auf die Datei "varset.h" verwiesen.

6.2.2 fwtools.h

Die Datei "fwtools.h" ist als eine Art Werkzeugkasten gedacht: wichtige Funktionen, die immer wieder bei einer Analyse, besonders bei einer Transferfunktion benötigt werden, können hier abgelegt werden, z.B. die Funktionen

- `int fwtools_assignedVariable(ParseTree t)` (liefert, falls der übergebene Parsebaum eine Zuweisung repräsentiert, die Variable, der etwas zugewiesen wurde; sonst -1), und
- `int fwtools_VariableinExpression(int var, ParseTree t)` (liefert 1, falls eine Variable im übergebenen Parsebaum vorkommt).

Für weitere Dokumentation sei erneut auf den Quellcode hingewiesen ("fwtools.h").

Anhang A

Grammatik von *While*

Die Grammatik ist größtenteils dieselbe, die auch im Parsermodul "parser.y" verwendet wird. Das Nichtterminal <IOStatement> wird nur von `whrun` gebraucht bzw. abgeleitet, *wAnalyzer* unterstützt keine I/O-Befehle. Zur Syntax von Labels, Variablen und Integerkonstanten siehe Kap. 3.

<Prog> ::= <Statement> `;` <Prog> |
<Statement>

<Statement> ::= <Label> <SimpleStatement> |
<SimpleStatement>

<SimpleStatement> ::= <SkipStatement> |
<IfStatement> |
<WhileStatement> |
<AssignStatement> |
<IOStatement>

<SkipStatement> ::= `SKIP`

<IfStatement> ::= `IF` <BoolExpr> `THEN` <Prog> `ELSE` <Prog> `FI`

<WhileStatement> ::= `WHILE` <BoolExpr> `DO` <Prog> `OD`

<AssignStatement> ::= <Variable> `:=` <ArithExpr>

<IOStatement> ::= `GET`(<Variable>) |
`PUT`(<Variable>)

```
<BoolExpr> ::= TRUE |
               FALSE |
               <RelExpr> |
               <BoolExpr> AND <BoolExpr> |
               <BoolExpr> OR <BoolExpr> |
               NOT <BoolExpr> | (<BoolExpr>)

<RelExpr> ::= <ArithExpr> = <ArithExpr> |
              <ArithExpr> <> <ArithExpr> |
              <ArithExpr> <= <ArithExpr> |
              <ArithExpr> < <ArithExpr> |
              <ArithExpr> >= <ArithExpr> |
              <ArithExpr> > <ArithExpr>

<ArithExpr> ::= <ArithExpr> + <ArithExpr> |
                <ArithExpr> - <ArithExpr> |
                <ArithExpr> * <ArithExpr> |
                <ArithExpr> DIV <ArithExpr> |
                <ArithExpr> MOD <ArithExpr> |
                <ArithExpr> POT <ArithExpr> |
                <ArithExpr> EXPT <ArithExpr> |
                - <ArithExpr> |
                (<ArithExpr>) | <Integer> | <Variable>
```

Anhang B

Implementierte Analysearten

In *wAnalyzer* sind standardmäßig einige Analysearten integriert, die hier der Vollständigkeit halber erläutert werden sollen. Insbesondere werden hier die Transferfunktionen der Analysen informell beschrieben; für genauere Definitionen sei auf [Kö03] und [NNH99] verwiesen. Einer Transferfunktionen wird ein Knoten des Flussgraphen und dessen (aktueller) Analysewert (sozusagen der Wert am Eingang des Knotens) als Parameter übergeben; daraus errechnet die Funktion einen neuen Analysewert (den Ausgabewert, der ausdrückt, was mit dem Analyseergebnis beim Passieren des Knotens geschieht

- "Simple Analysis" ermittelt für jeden Knoten im Flussgraphen den kürzesten Programmpfad, der benötigt wird, um den mit dem Knoten korrespondierenden Programmblock zu erreichen. Die der Instanz zugrunde liegende Struktur wird auch als "tropischer Semiring" bezeichnet. Die Transferfunktion besteht aus einer einfachen Inkrementierung des Analysewertes (das bedeutet also, dass ein weiterer Schritt im Programm zurückgelegt wird). Die Analyse ist deshalb eine Vorwärtsanalyse.

- "Lebendige Variablen" ermittelt, wie der Name schon sagt, für jeden Knoten die Menge der lebendigen Variablen. Eine Variable heisst *lebendig* am Ausgang eines Programmblocks, wenn es einen Pfad von diesem Block zu einem anderen Block gibt, der diese Varable in einer Bedingung oder auf der rechten Seite einer Zuweisung benutzt, ohne dass die Variable vorher neu definiert wird. Grundmenge des Frameworks ist daher die Potenzmenge der Menge der Programmvariablen. Daraus ergibt sich auch die dazugehörige Transferfunktion: Repräsentiert der jeweilige Knoten eine Zuweisung, so wird die dadurch veränderte Variable aus der Analysemenge entfernt und die im arithmetischen Ausdruck rechts der Zuweisung stehenden Variablen eingefügt. Repräsentiert der

Knoten einen booleschen Ausdruck (`IF`, `WHILE`), so werden die im Ausdruck vorkommenden Variablen der Analysemenge hinzugefügt. Demgemäß muss der Worklistalgorithmus von unten nach oben arbeiten, also ist "Lebendige Variablen" eine Rückwärtsanalyse. Es ist noch anzumerken, dass zu Beginn die Initialisierungswerte der Endknoten des Programms vom Benutzer abgefragt werden müssen; diese entsprechen Ausgabevariablen des Programms.

- "Stark lebendige Variablen": Diese Analyse ähnelt sehr der Analyse "Lebendige Variablen", jedoch benutzt sie eine etwas modifizierte Transferfunktion: Repräsentiert der übergebene Knoten eine Zuweisung, und befindet sich in der Eingangsanalysemenge die Zuweisungsvariable nicht, wird einfach die Eingangsanalysemenge als Rückgabewert der Transferfunktion verwendet, da die Variable selbst ja an diesem Punkt nicht lebendig ist und somit keine Veranlassung besteht, die in der Zuweisung noch vorkommenden Variablen in die Analyse einzubeziehen.

- "Initialisierung von Variablen" ermittelt für jeden Programmblock die Menge der Variablen, die *noch nicht* initialisiert wurden, d.h. diejenigen, denen noch kein Wert zugewiesen wurde. Auch hier dient uns als Grundmenge die Potenzmenge der Variablenmenge. In der Transferfunktion werden nur Zuweisungen betrachtet; bei allen anderen Anweissungen wird die Eingabeanalysemenge zur Ausgabeanalysemenge. Liegt jedoch eine Zuweisung vor, so wird überprüft, ob der veränderten Variablen ein Ausdruck zugewiesen wird, in dem eine uninitialisierte Variable vorkommt. Ist dies der Fall, ist auch die veränderte Variable möglicherweise uninitialisert und muss in die Ausgangsanalysemenge aufgenommen werden; gibt es in dem Ausdruck keine uninitialiserten Variablen, ist die Ausgangsanalysemenge gleich der Eingangsanalysemenge ohne die veränderte Variable. Es liegt hier demnach eine Vorwärtsanalyse vor.

	(Stark) Lebendige Variablen	Initialisierung von Variablen	Simple Analysis
L	Potenzmenge der Variablenmenge	Potenzmenge der Varaiblenmenge	$\mathbb{N} \cup \infty$
$\sqcup$ (Supremum)	$\cup$	$\cup$	min
$\perp$ (Bottom)	$\emptyset$	$\emptyset$	∞
E (Extremallabel)	Endknoten des Programms	Anfangsknoten des Programms	Anfangsknoten des Programms
Richtung	Rückwärtsanalyse	Vorwärtsanalyse	Vorwärtsanalyse
ι	Benutzerspezifisch	Menge aller Variablen	1

Literaturverzeichnis

[Ise01] Rolf Isernhagen. *Softwaretechnik in C und C++*. Carl Hanser Verlag München, 2001.

[Kö03] Barbara König. *Skript zur Vorlesung Programmanalyse*. Universität Stuttgart, 2003.

[LMB92] John Levine, Tony Mason, and Doug Brown. *lex & yacc, 2nd Edition*. O'Reilly, 1992.

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

[Wil95] Gerhard Willms. *Das C-Grundlagen-Buch*. Data Becker, 1995.