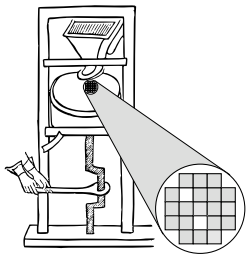
Technische Universität Darmstadt
FG Eingebettete Systeme und
ihre Anwendungen (ESA)

Prof. Dr. Andreas Koch



"Optimierende Compiler"
Aufgabe 2: Whole-Program-Analyse
Abgabe bis zum 31.05.2007, 18:00 Uhr MET DST

Um einen Einblick in die Analyse von Zusammenhängen über komplette Programme hinweg zu bekommen, realisieren Sie in dieser Phase einen Pass, der die in allen Anweisungen gelesenen und geschriebenen Variablen bestimmt. Diese Angaben dienen als Grundlage für die Arbeiten der nächsten Phasen.

1 Einleitung

In vielen weiteren Optimierungsschritten ist ein wesentliches Kriterium, welche Variablen tatsächlich in einer Anweisung benutzt werden. Dabei ist zu unterscheiden zwischen dem *Lesen* und *Schreiben* einer Variablen. Das Lesen findet beispielsweise beim Auftreten einer Variablen im Ausdruck auf der rechten Seite einer Zuweisung bzw. bei der Übergabe als Wertparameter (ohne var) an eine Prozedur oder Funktion statt. Beispiele für das Schreiben von Variablen sind ihre Verwendung auf der linken Seite einer Zuweisung und die Übergabe als var-Parameter an eine Prozedur.

Diese Angaben können nun auch über größere Teile des Eingabeprogrammes als eine einzelne Anweisung gesammelt werden. Die in einer Anweisungs*folge* gelesenen und geschriebenen Variablen sind beispielsweise die Vereinigungsmengen der in den einzelnen Anweisungen gelesenen und geschrieben Variablen. Dabei findet hier eine rein *statische* Analyse statt: Bei einem if-then-else-Konstrukt werden die in *beiden* Ästen auftretenden Variablen als benutzt betrachtet.

Komplizierter wird es, wenn die Analyse einen noch größeres Betrachtungsrahmen wählt: Bei der Analyse über Prozedurgrenzen hinweg wird an jeder Aufrufstelle (die ja selbst eine Anweisung ist) die Benutzung der Variablen ermittelt. Auf den ersten Blick scheint es ausreichend, solche Variablen als "gelesen" zu vermerken, die *ohne* var übergeben werden, und die var-Parameter (Übergabe mittels Referenz) als "geschrieben" anzusehen.

Die Realität ist komplexer, da eine Prozedur ja auch noch *globale* und (im Fall von Triangle) auch noch *nicht-lokale* Variablen verwenden kann. Es muss also an der Aufrufstelle auch noch in die Prozedur *hineingesehen* werden, um die korrekten Informationen zu ermitteln.

Triangle-Programme sind ausreichend einfach, dass eine solche verfeinerte Analyse über das *gesamte* Programm (*whole-program analysis*) durchgeführt werden kann.

2 Problemstellung

2.1 Analyse

Erweitern Sie die interaktive Oberfläche des Triangle-Compilers um ein Kommando `genrwset`, das zu jeder Anweisung des eingelesenen und kontextanalysierten Triangle-Programmes die Mengen der gelesenen und geschriebenen Variablen bestimmt.

Bei nicht-primitiven Variablen (Array, Record, etc.) soll nur die Basisvariable betrachtet werden. Das heisst, dass lesende Zugriffe auf Array-Elemente A(1) und A(2) behandelt werden als Lese-Zugriffe auf das Array A. Analog werden Zugriffe auf Teile eines Records als Zugriffe auf den Record selber angelegt: date.m und date.d werden vermerkt als Zugriffe auf date.

Diese statische Analyse soll dabei *über das gesamte Programm* erfolgen. Die Analyse muss dabei mit den Blättern der Programmhierarchie beginnen. Dabei handelt es sich um Prozeduren/Funktionen, die selber keine weiteren Prozeduren/Funktionen mehr aufrufen. Hinweis: Sie müssen dabei *direkte* Rekursion behandeln (eine Prozedur ruft sich selbst wieder auf), die korrekte Verarbeitung auch indirekter Rekursion ist *optional* (kann aber zur Aufwertung Ihrer Abgabe führen). Desweiteren müssen Sie die in Triangle eingebauten Basisprozeduren und Funktionen (z.B. getint(var zahl) korrekt erkennen und bearbeiten.

Das Kommando `genrwset` soll selber im Normalbetrieb keine Ausgabe liefern, sondern seine Ergebnisse direkt in den von Ihnen geeignet erweiterten AST annotieren (Sie können aber gerne Kommandozeilenparameter für einen Debug- oder Verbose-Modus einbauen).

Zur weiteren Dekoration des DASTs soll die Klasse `Command` um zwei Instanzvariablen `reads` und `writes` erweitert werden, die zu jedem `Command` die Analyse-Ergebnisse in Form von Verweisen auf die `Declaration` der Variable abspeichern. Welchen Datentypen Sie dafür konkret wählen bleibt Ihnen überlassen. Eine Möglichkeit wären aber beispielsweise Listen aus `Declaration`-Objekten.

Durch das Verweisen auf die `Declaration`, die ja die Wurzel des eigentlichen Deklarationsunterbaumes im DAST darstellt, ist auch die eindeutige Identifizierbarkeit der Variablen sichergestellt: Namen könnten ja in unterschiedlichen Geltungsbereichen mehrfach auftauchen und wären daher für diese Anwendung ungeeignet.

2.2 Ausgabe der Ergebnisse

Für das Debugging ist es sinnvoll, das Endergebnis in einem festen Ausgabeformat schreiben zu können. Gehen Sie dafür hier wie folgt vor: Erweitern Sie die in der Vorphase enstandenen Kommandos `dumpast` und `showast`, dass diese bei vorherigem Ausführen von `genrwset` auch diese neu bestimmten Daten mit anzeigen.

Soweit noch nicht geschehen erweitern Sie die Kommandos auch so, dass bei Angabe eines auf das Kommando folgenden Strings die Ausgabe nicht auf die Konsole stattfindet, sondern in eine Datei geschrieben wird. Beispielsweise soll durch `showast test.ast` die Ausgabe in die Datei `test.ast` erfolgen. Diese Funktionalität wird sich auch bei der Erstellung der für die Abgabe unten geforderten Dateien als nützlich erweisen.

2.2.1 Erweiterung von `showast`

`showast` soll die Ergebnisse der `genrwset`-Analyse an passender Stelle als *Kommentare* in das ausgegebene Triangle-Programm einfügen. Auch hier muss der ausgegebene Triangle-Quelltext hinterher wieder korrekt kompilierbar sein! Da, wie oben bereits angedeutet, der Variablenname alleine für die eindeutige Identifizierung einer Variable nicht ausreichend ist, werden (nur zu Darstellungsgründen) den Variablen auch noch über gesamte Programm eindeutige Nummern zugeordnet. Diese Angaben

```
let
  ! genrwset: decl {x.0}
  var x : Integer;
  ! genrwset: decl {y.1}
  var y : Integer;

  ! genrwset: decl {k.2, l.3} read {k.2,l.3,y.1} write {k.2,y.1}
  proc foo (var k : Integer, l : Integer) ~
    let
      ! genrwset: decl {o.4}
      var o : Integer;

      ! genrwset: decl {k.5,l.6} read {k.5,l.6,y.1} write {k.5,y.1}
      proc bar (var k : Integer, l : Integer) ~
        let
          ! genrwset: decl {dummy.7}
          var dummy : Integer
        in begin
          ! genrwset: read {l.6,y.1} write {k.5}
          k := l * y;
          ! genrwset: read {k.5} write {y.1}
          y := k + 1
        end

    in begin
      ! genrwset: read {l.3} write {o.3}
      o := l * 2;
      ! genrwset: read (k.2,o.3,y.1} write {k.2,y.1}
      bar(var k, o)
    end


in begin
  ! genrwset: write {x.0}
  x := 42;
  ! genrwset: read {x.0,y.1} write {x.0,y.1}
  foo(var x, 3);
  ! genrwset: read {x.0}
  putint(x);
  ! genrwset: read {y.1}
  putint(y)
end
```

Abbildung 1: Beispielausgabe von `showast` nach `genrwset`

können Sie in einer Instanzvariable `idnum` in `Declaration` abspeichern. **Wichtig**: Diese Variablennummern dienen *nur* zum Identifizieren von unterschiedlichen Variablen gleichen Namens in der *Ausgabe*. Für die Abläufe innerhalb des Compilers liegt ja bereits im DAST der *immer eindeutige* Verweis auf die Wurzel des entsprechenden Deklarations-Unterbaumes vor.

Abbildung 1 illustriert sowohl die Arbeitsweise des Algorithmus auch als das gewünschte Ausgabeformat. Alle relevanten Zeilen (Deklarationen, Anweisungen) bekommen einen ! genrwset: Kommentar vorangestellt. In diesem werden in drei Kategorien decl, read, write die jeweils deklarierten, gelesenen und geschriebenen Variablen angegeben. Variablen werden dabei identifiziert durch ihren Namen, einem Punkt, sowie ihrer eindeutigen (bei der Deklaration vergebenen) Identifikationsnummer.

Einige Hinweise zu den Daten in der Abbildung: Beachten Sie, dass `bar` auch die globale Variable y.1 liest und schreibt. Bei dieser Art der Analyse wird auch ermittelt, dass hier auch k.5 gelesen wird. Das ist zwar nicht ganz richtig (der alte Wert der als k.5 übergebenen Variable wird nie gebraucht), solche Arten von Unterscheidungen braucht Ihr Algorithmus aber nicht vorzunehmen (kommt später → Datenflußanalyse). Nicht alle gelesenen formalen Parameter einer Prozedur müssen an der Aufrufstelle

```
(AssignCmd
  V: ...
  E: ...
  reads: {
            (VarDeclaration idnum: 7
              I:(Identifier spelling:"a"
              )
              T:(IntTypeDenoter
              )
             )
            (VarDeclaration idnum: 14
              I:(Identifier spelling:"b"
              )
              T:(IntTypeDenoter
            )
          }
  writes: {
            (VarDeclaration idnum: 12
              I:(Identifier spelling:"y"
              )
              T:(IntTypeDenoter
              )
            )
           }
)
```

Abbildung 2: Verkürzte Beispielausgabe von `dumpast` von y := a+b

der Prozedur tatsächlich zu gelesenen aktuellen Parametern führen. Beispielsweise liest foo auch l.3. Da aber beim Aufruf für diesen formalen Parameter als aktueller Parameter eine Konstante übergeben wird (der Wert 3), wird an der Aufrufstelle dafür keine zu lesende Variable vermerkt.

2.2.2 Erweiterung von `dumpast`

`dumpast` soll, auf gleicher Ebene wie die `V` und `E`-Angaben, nun auch die Inhalte der `reads` und `writes`-Variablen sowie darin auch ihre Identifikationsnummern ausgeben. Die Ausgaben für eine Zuweisungsanweisung y := a + b könnte also wie in Abbildung 2 gezeigt aussehen. Hier wurde angenommen, dass die Variablen a, b, y vorher noch die eindeutigen Identifikationsnummern 7, 14 und 12 zugeordnet bekommen haben. Beachten Sie: Die Listen können durchaus auch *leer* sein. Z.B. würde a := 1 keine Variablen lesen, und der Prozeduraufruf putint(42) weder Variablen lesen noch schreiben.

3 Abgabe

Es gelten auch hier die auf dem ersten Aufgabenblatt beschriebenen Anforderungen an **Programmierstil** und **Dokumentation**. Bitte lesen Sie die entsprechenden Abschnitte falls nötig nocheinmal!

Jede Gruppe schickt spätestens zum Abgabezeitpunkt in einem `.jar`-Archiv alle Dateien ihrer Version des Triangle-Compilers an

`oc07@esa.informatik.tu-darmstadt.de`

mit dem Subject `Abgabe 2 Gruppe` *N*, wobei Ihnen *N* bereits in der Vorlesung mitgeteilt wurde (falls vergessen: siehe nächster Abschnitt). In dem Archiv sollen nicht nur die eigenen, sondern *alle* (auch unmodifizierten) Quellen des Triangle-Compilers enthalten sein. Ebenso legen Sie eventuell verwendete zusätzliche externe Bibliotheken in Form ihrer jeweiligen `.jar`-Dateien bei (aber siehe Abschnitt 6).

Gruppe	Mitglieder	Zeit
1	Güntner/Wodniok/Welti	16:15-16:30
2	Pottharst/Vogel/Bräckelmann	16:30-16:45
3	Huff/Rendel/Degenhardt	16:45-17:00
4	Todorov/Roth/Mihaylov	17:00-17:15
5	Wasser/Moldenhauer	17:15:17:30

Tabelle 1: Gruppennummern und Kolloquiumzeiten

Neben den Java-Quelltexten umfasst das Abgabearchiv eine Datei `README.txt`, die enthält:

- die Namen der Gruppenmitglieder.
- eine Übersicht über die neuen und geänderten Dateien mit jeweils einer kurzen (eine Zeile reicht) Beschreibung ihrer Funktion.
- Hinweise zur Compilierung der Quellen. Geben Sie eine `javac`-Kommandozeile an bzw. verweisen Sie auf mitgelieferte Makefiles oder ANT Build-Dateien. *Nicht* ausreichend ist ein Hinweis auf eine von Ihnen verwendete IDE (wie Eclipse, NetBeans etc.).
- Angaben über weitere Bibliotheken (beispielsweise JSAP, log4j, JUnit etc.), die Sie eventuell verwendet haben. Diese Bibliotheken legen Sie bitte dann auch als .jar Dateien in das abgegebene Archiv.
- für alle Beispielprogramme die Ausgaben von `showast` und `dumpast` nach `read` / `check` / `genrwset`

4 Beurteilung

In dieser Phase beginnen die Kolloquien. Sie finden am 19.6.07 in der regulären Vorlesungszeit in Raum E103 statt. Diese Kolloquien sind Bestandteil der Prüfungsleistung, es besteht daher **Anwesenheitspflicht** für alle Gruppenmitlglieder.

Ein Kolloquium dauert jeweils ca. 15 Minuten. Die Anfangszeiten (und falls nötig Ihre Gruppennummer) entnehmen Sie bitte Tabelle 1.

5 Gruppenarbeit und Hinweise

Der Analyseteil hat sicherlich einen höheren Arbeitsaufwand als der Ausgabeteil, der ja nur die von der Analyse erzeugten Daten anzeigt.

Auch bei den Tests von Analysen ist es häufig so, dass auf den ersten Blick alles funktioniert. Aber dann in einem Quelltext ein "Sonderfall" auftaucht, der vom bisherigen Analyse-Code noch nicht oder nur fehlerhaft bearbeitet wird. Beachten Sie insbesondere, dass Ihr Programm nun mit dem vollen Triangle-Sprachumfang funktionieren muss, nicht mehr nur mit der Mini-Triangle-Untermenge aus der Vorlesung.

Unterschätzen Sie daher keinesfalls den Aufwand für die Erstellung von Testprogrammen und den sorgfältigen Test der Analyseergebnisse. Zwar wurde Ihnen auf dem Web-Site der Veranstaltung eine ganze Reihe von Beispielprogrammen in Triangle zur Verfügung gestellt. Diese sind aber bei der Entwicklung und dem Debugging der `genrwset`-Analyse schon viel zu kompliziert. Sie sollten daher

zunächst eine Sammlung von *kürzeren* Programmen entwickeln, die (hoffentlich) alle für *diese* Analyse relevanten Fälle abdecken.

Falls in Ihrer Gruppe eine Situation entstehen sollte, in der einzelne Mitglieder deutlich zuwenig (oder zuviel!) der anfallenden Arbeitslast bewältigen, sprechen Sie den Betreuer bitte *frühzeitig* auf die Problematik an. Nur so kann durch geeignete Maßnahmen in Ihrem Interesse gegengesteuert werden. Nach der Abgabe ist es dafür **zu spät** und Sie tragen die Konsequenzen selber (z.B. wenn sich eines Ihrer Team-Mitglieder wegen seiner Verpflichtungen beim Wasser-Polo nur stark eingeschränkt den Mühen der Programmierung widmen konnte, und Sie daher eine unvollständige Lösung abgeben mussten).

6 Plagiarismus

Der Fachbereich Informatik misst der Einhaltung der Grundregeln der wissenschaftlichen Ethik großen Wert bei. Zu diesen gehört auch die strikte Verfolgung von Plagiarismus. Mit der Abgabe einer Lösung zu den Programmierprojekten bestätigen Sie, dass Ihre Gruppe die alleinigen Autoren des neuen Materials bzw. der Änderungen des zur Verfügung gestellten Codes sind. Im Rahmen dieser Veranstaltung dürfen Sie den Code des Triangle-Compilers vom OC07 Web-Site sowie Code-Bibliotheken für nebensächliche Programmfunktionen frei verwenden. Mit anderen Gruppen dürfen Sie sich gerne über grundlegende Fragen zur Aufgabenstellung austauschen. Detaillierte Lösungsideen dürfen dagegen *nicht vor Abgabe*, Artefakte wie Programm-Code oder Dokumentationsteile *überhaupt nicht* ausgetauscht werden. Bei Unklarheiten zu diesem Thema (z.B. der Verwendung weiterer Software-Tools oder Bibliotheken) sprechen Sie bitte Ihren Betreuer gezielt an.