Datenflussanalyse mit SATIrE: 'Live Variables'

Gergö Barany, gergo@complang.tuwien.ac.at

VU Optimierende Übersetzer, WS 2014/2015

Dieses Dokument bietet anhand eines Beispiels eine Einführung in die Entwicklung von Datenflussanalysen mit SATIrE. In elektronischer Form ist das Dokument auf der Übungsmaschine `g0` unter `/usr/local/optub/doc/live_variables.pdf` zu finden, der Quellcode der hier behandelten Analysespezifikation liegt unter `/usr/local/optub/share/live_variables.optla`.

1 Die Analyse: Live Variables

Eine Programmvariable ist an einem Programmpunkt *lebendig* (engl. *live*), wenn sie einen Wert enthält, der irgendwann in der Zukunft ausgelesen werden könnte. Andernfalls ist sie *tot* (*dead*). Die 'Live Variables'-Analyse soll für jeden Programmpunkt feststellen, welche Variablen lebendig sein könnten. Diese Information kann etwa verwendet werden, um überflüssige Zuweisungen aus einem Programm zu entfernen. Auch für die Registerallokation ist Information über die Lebensdauer von Variablen nützlich.

2 Datenflussanalysen mit PAG und SATIrE

Datenflussanalysen können oft durch einfache Gleichungssysteme beschrieben werden. Für diese Systeme kann mittels Fixpunktsuche eine Lösung gefunden werden. Um den Schritt von der Gleichungsdarstellung zu einer vollständigen Analyseimplementierung zu automatisieren, verwenden wir den 'Programmanalysegenerator' PAG, welcher aus einer deklarativen Spezifikation eine Implementierung der Analyse in der Programmiersprache C erstellt.

Die von PAG generierte Analysebibliothek muss mit zusätzlichen Programmteilen verbunden werden, um ein vollständiges, lauffähiges Programm zu ergeben. Dazu gehören etwa ein Parser für das Eingabeprogramm sowie ein Programmteil zur Erstellung des Kontrollflussgraphen, auf dem PAG operiert. Nach der Datenflussanalyse können ihre Ergebnisse auf unterschiedliche Weise visualisiert oder als Annotationen in das Programm eingefügt werden. Für all diese Aufgaben werden vom SATIrE-System Module bereitgestellt.

3 Beispiel: Analysespezifikation für 'Live Variables'

Dieser Abschnitt enthält den ausführlich kommentierten Quelltext einer Spezifikation der 'Live Variables'-Analyse mittels PAG und der von SATIrE zur Verfügung gestellten Infrastruktur. Die Analyse behandelt eine kleine Teilmenge der Programmiersprache C. In dieser Teilsprache kommen Verzweigungen und Schleifen vor, aber keine Funktionsaufrufe oder Zeigerausdrücke. Seiteneffekte in Ausdrücken mit den Operatoren `++` und `--` sind hingegen zulässig.

Jede Analysespezifikation beginnt mit der Deklaration der von ihr verwendeten Datentypen. Wir schreiben Typen in dieselbe Datei wie die eigentliche Analysespezifikation. Ältere Dokumente zu PAG und SATIrE sprechen noch oft von einer Trennung in `.set`- und `.optla`-Datei für diese zwei Teile.

Typen werden mittels Typkonstruktoren wie `set` (Menge), `list` (Liste) oder `->` (endliche Abbildung) definiert. Die rechte Seite jeder Definition muss genau einen Konstruktor enthalten, verschiedene Definitionen mit gleichen rechten Seiten sind unzulässig. Der `SET`-Abschnitt enthält Definitionen beliebiger Typen, im `DOMAIN`-Abschnitt müssen Verbände definiert werden. Die Datenflussinformation jeder Analyse muss ein Verband sein.

```
SET

DOMAIN
  VariableSet
      = set(VariableId)
  VariableLattice
      = lift(VariableSet)
ENDDOMAIN
```

Für den Fall der 'Live Variables'-Analyse soll die Analyseinformation für jeden Programmpunkt aus einer Menge von Variablen bestehen; es soll auch einen von der leeren Menge verschiedenen 'undefinierten' Wert für die Information an noch nicht besuchten Programmpunkten geben.
Ausgehend vom vordefinierten Typ `VariableId` für eindeutige Kennungen von Programmvariablen wird daher zunächst der Typ `VariableSet` definiert. Mit dem Typkonstruktor `lift` werden zwei ausgezeichnete Werte $\top$ (`top`) und $\bot$ (`bot`) hizugefügt, die bei Vergleichen als größer bzw. kleiner als jede Variablenmenge gelten. Damit kann `bot` als undefinierter Analysewert verwendet werden.

```
PROBLEM Live_Variables
  direction:  backward
  carrier:    VariableLattice
  init_start: lift({})
  init:       bot
```

Auf die Typdeklarationen folgt ein Block, der allgemeine Parameter der Analyse festlegt. 'Live Variables' ist eine Rückwärtsanalyse auf dem oben definierten Typ `VariableLattice`. Am Ende des Programms sind alle Variablen tot, daher ist der Initialisierungswert für den Startpunkt der Analyse (den Endknoten des Kontrollflussgraphen) die leere Variablenmenge. Alle anderen Knoten werden mit dem Wert `bot` initialisiert, um anzuzeigen, daß sie noch nicht besucht wurden.

```
  equal:      varlattice_equal
  combine:    varlattice_combine
  retfunc:    varlattice_combine
  widening:   varlattice_combine
```

Schließlich werden die Funktionen benannt, die für die Überprüfung von Datenflusswerten auf Gleichheit sowie für die Kombination von Datenflussinformation von unterschiedlichen Programmpfaden verantwortlich sind. Dabei ist `combine` die allgemeine Kombinationsfunktion; `retfunc` und `widening` sind spezielle Varianten, die hier nicht benötigt werden. Die Definitionen von Hilfsfunktionen befinden sich weiter unten im Abschnitt `SUPPORT`.

`TRANSFER`

Im Abschnitt `TRANSFER` werden die Transferfunktionen für die verschiedenen Arten von Anweisungen definiert. Der Kopf einer derartigen Definition besteht aus einem Anweisungsmuster und einem Kantentyp. Das Muster für die Anweisung beschreibt abstrakte Syntaxbäume durch Konstruktoren und Variablen. Entspricht die Struktur einer Anweisung im Programm dem Muster, dann werden die Variablen im Muster entsprechend an Teilbäume gebunden. Der Kantentyp kann etwa verwendet werden, um bei Verzweigungen im Programm unterschiedliche Transferfunktionen für die Fälle 'Bedingung wahr' und 'Bedingung falsch' anzugeben. Für 'Live Variables' sind Kantentypen nicht relevant, daher kann einfach die anonyme Variable `_` als Kantentyp angegeben werden.
Der Körper einer Transferfunktion ist ein Ausdruck, der Hilfsfunktionen verwenden kann. Der Wert des Ausdrucks ist die neue Datenflussinformation für den Punkt nach (bei Rückwärtsanalysen: vor) der untersuchten Anweisung; die Datenflussinformation am Eingang dieser Anweisung ist durch die spezielle Variable `@` gegeben.

```
ExprStatement(expr), _:
  let live_vars <= @;
  in  lift(transfer(expr, live_vars));

ScopeStatement(WhileStmt(ExprStatement(expr))), _:
  let live_vars <= @;
  in  lift(transfer(expr, live_vars));

ScopeStatement(IfStmt(ExprStatement(expr))), _:
  let live_vars <= @;
  in  lift(transfer(expr, live_vars));
```

In der 'Live Variables'-Analyse sollen primär drei Arten von Anweisungen behandelt werden: Anweisungen, die ausschließlich aus einem Ausdruck (etwa einer Zuweisung) bestehen, außerdem Schleifen sowie Verzweigungen. In all diesen Fällen muß ein Ausdruck darauf geprüft werden, welche Variablen gelesen bzw. geschrieben werden. Diese Aufgabe ist in die weiter unten definierte Hilfsfunktion `transfer` ausgelagert.

Der Ausdruck `let` v `<=` e_1 `in` e_2 kann angewendet werden, um einen Wert aus dem Ausdruck e_1 von einem Typ `lift(`τ`)` 'auszupacken': Hat e_1 den Wert `top` oder `bot`, so ist das der Wert des gesamten

Ausdrucks. Andernfalls wird die Variable v an den 'inneren' Wert vom Typ τ gebunden und e_2 mit dieser Bindung ausgewertet. In den ersten drei Transferfunktionen sind also das zweite Argument und der Rückgabewert der Hilfsfunktion `transfer` jeweils vom 'einfachen' Typ `VariableSet`. Der `let`-Ausdruck mit `<=`-Operator ermittelt diesen Wert aus der Datenflussinformation `@`, die Funktion `lift` 'hebt' das Ergebnis wieder in den Typ `VariableLattice`. Meist gilt bei Analysen mit einem Typ `lift(`τ`)` für die Datenflussinformation, dass die Transferfunktionen die Form `let` v `<= @; in lift(...)` haben.

```
WhileJoin(), _: @;
IfJoin(), _: @;
```

Die Anweisungen `WhileJoin` und `IfJoin` markieren die Stellen, an denen der Kontrollfluss am Ende einer Schleife oder Verzweigung zusammenfließt. Sie sind für die 'Live Variables'-Analyse nicht von Belang, da sie keine Variablen verwenden oder definieren. Daher wird die Eingangsinformation `@` unverändert weitergegeben.

```
DeclareStmt(var_symbol, type), _:
  let live_vars <= @;
      var = varsym_varid(var_symbol);
  in  lift(
          if  var ? live_vars  then
            println("warning: '", varid_str(var),
                    "' may be used uninitialized")
            live_vars # var
          else
            live_vars
          );
```

Die Anweisung `DeclareStmt` steht für die Deklaration einer Variablen, das erste Argument ist das entsprechende Variablensymbol. Vor der Deklaration kann eine Variable bestimmt nicht verwendet werden, daher soll sie in der Datenflussinformation, die nach oben weitergereicht wird, nicht mehr vorkommen. Die mittels `varsym_varid` zum Variablensymbol ermittelte Kennung wird daher mit dem `#`-Operator aus der Menge der lebendigen Variablen entfernt. Zusätzlich kann an dieser Stelle eine Warnung ausgegeben werden, wenn die deklarierte Variable lebendig ist, denn dies deutet auf eine Verwendung der Variablen ohne vorhergehende Initialisierung hin.

```
UndeclareStmt(variables), _: @;
```

Der Anweisungstyp `UndeclareStmt` bezeichnet Programmpunkte, an denen der Gültigkeitsbereich von Variablen endet. Diese Punkte sind für die vorliegende Analyse irrelevant.

```
FunctionEntry(name), _: @;
FunctionExit(name, local_vars), _: @;
```

Knoten der Typen `FunctionEntry` und `FunctionExit` bezeichnen Anfangs- und Endpunkt einer Funktion. Für die 'Live Variables'-Analyse sind sie nicht von Interesse.

```
statement, _:
  println("warning: cannot handle ",
          "statement of the form: ",
          stmt_asttext(statement))
  @;
```

Die letzte Transferfunktion wird ausgeführt, falls keines der obigen Muster auf die untersuchte Anweisung passt. Die Analyse ermittelt hier mit Hilfe der vordefinierten Hilfsfunktion `stmt_asttext` eine textuelle Darstellung der Struktur der Anweisung und gibt diese in einer Warnung an den Benutzer aus. Als Datenflussinformation wird einfach die Eingangsinformation weitergegeben; strenggenommen müßte die Analyse abgebrochen werden, wenn eine Anweisung mit unbekannter Semantik angetroffen wird.

```
SUPPORT
```

Der Abschnitt `SUPPORT` enthält Definitionen von Hilfsfunktionen, die aus den Transferfunktionen aufgerufen werden. PAG ist meist in der Lage, den Typ einer Funktion automatisch zu ermitteln, Typannotationen sind daher nicht zwingend vorgeschrieben.

```
transfer :: Expression, VariableSet -> VariableSet;
transfer(expression, live_vars) =
  let kill_set = var_defs(expression);
      gen_set  = var_uses(expression);
  in  union(subtract(live_vars, kill_set), gen_set);
```

Die Funktion `transfer` berechnet, ausgehend von einer Menge von lebendigen Variablen und einem Ausdruck, eine neue Menge von Variablen, die unmittelbar vor Auswertung des Ausdrucks lebendig sein könnten. Dies entspricht der Berechnung der Menge $\mathsf{LV}_\bullet = (\mathsf{LV}_\circ \setminus K) \cup G$, wobei $\mathsf{LV}_\bullet$ und $\mathsf{LV}_\circ$ die Mengen der lebendigen Variablen vor bzw. nach Auswertung des Ausdrucks und K und G die Mengen der vom Ausdruck überschriebenen bzw. gelesenen Variablen bezeichnen.

```
var_defs :: Expression -> VariableSet;
var_defs(expr) =
  case  expr  of
    IntVal(_)             => {};
    VarRefExp(_)          => {};

    PlusPlusOp(VarRefExp(_) as varref)
                          => {varref_varid(varref)};
    MinusMinusOp(VarRefExp(_) as varref)
                          => {varref_varid(varref)};

    AddOp(lhs, rhs)       => union(var_defs(lhs),
                                   var_defs(rhs));
    SubtractOp(lhs, rhs) => union(var_defs(lhs),
                                   var_defs(rhs));
    MultiplyOp(lhs, rhs) => union(var_defs(lhs),
                                   var_defs(rhs));
    DivideOp(lhs, rhs)   => union(var_defs(lhs),
                                   var_defs(rhs));

    LessThanOp(lhs, rhs)                 => union(var_defs(lhs), var_defs(rhs));
    GreaterThanOp(lhs, rhs)              => union(var_defs(lhs), var_defs(rhs));

    AssignOp(VarRefExp(_) as varref, rhs) => union({varref_varid(varref)}, var_defs(rhs));

    _   => println("warning: unsupported expression in var_defs: ", expr_asttext(expr))
           {};
  endcase;
```

Die Funktion `var_defs` berechnet für einen Ausdruck die Menge jener Variablen, in die durch den Ausdruck geschrieben werden kann. Dies sind für die vorliegende Analyse jene, die die linke Seite einer Zuweisung bilden oder als Operand von `++` oder `--` vorkommen. Das `case`-Konstrukt zerlegt den Ausdruck, Teilausdrücke können an Variablen gebunden werden.
Da auch Ausdrücke mit Seiteneffekten geschachtelt werden können, muß `var_defs` rekursiv die gesamte Struktur des Ausdrucks betrachten.

```
var_uses :: Expression -> VariableSet;
var_uses(expr) =
  case  expr  of
    IntVal(_)           => {};
    VarRefExp(_)        => {varref_varid(expr)};

    AssignOp(_, rhs)    => var_uses(rhs);

    _   => if  is_unary(expr)  then
             var_uses(unary_get_child(expr))
           else if  is_binary(expr)  then
             union(var_uses(binary_get_left_child(expr)),
                   var_uses(binary_get_right_child(expr)))
           else
             println("warning: unsupported expression ",
                     "in var_uses: ", expr_asttext(expr))
             {};
  endcase;
```

Da es eine große Anzahl von ein- und zweistelligen Operatoren gibt, von denen in der Analyse fast alle gleich behandelt werden müssen, wäre eine explizite Auflistung aller Möglichkeiten in einem `case` sehr umständlich. Deswegen stellt SATIrE Hilfsfunktionen zur Verfügung, um die Stelligkeit solcher Operatoren testen und auf ihre Kinder zugreifen zu können, ohne sich für den konkreten Knotentyp zu interessieren. Die Funktion `var_uses` sammelt die Menge aller in einem Ausdruck ausgelesenen Variablen mit Hilfe dieser Hilfsfunktionen auf.

```
subtract :: VariableSet, VariableSet -> VariableSet;
subtract(a, b) = { v  !!  v <-- a,  if !(v ? b) };
```

PAG stellt keine Funktion für Mengensubtraktion zur Verfügung, daher muss sie hier definiert werden. Der Ausdruck `{ v !! v <-- a, if !(v ? b) }` bezeichnet die Menge aller Elemente `v` der Menge `a` mit der Eigenschaft, daß `v` nicht in der Menge `b` vorkommt.

```
varlattice_equal :: VariableLattice, VariableLattice
                    -> bool;
varlattice_equal(a, b) = (a = b);

varlattice_combine :: VariableLattice, VariableLattice
                      -> VariableLattice;
varlattice_combine(a, b) = (a lub b);
```

Auch die am Anfang der Analyse deklarierte '`equal`'-Funktion für Überpfüfung der Gleichheit von Analysedaten sowie die '`combine`'-Funktion zur Verknüpfung der Analysedaten von unterschiedlichen Programmpfaden werden in diesem Abschnitt definiert. Gleichheit wird mittels PAGs `=`-Operator ermittelt, der Kombinationsoperator ist die kleinste obere Schranke (*least upper bound*, `lub`).

Damit ist die Analysespezifikation abgeschlossen. Der nächste Abschnitt beschreibt, wie aus der Spezifikation ein lauffähiges Programm erstellt und dieses getestet werden kann.

4 Erstellen und Testen des Analyseprogramms

Auf dem Übungsrechner `g0.complang.tuwien.ac.at` sind SATIrE und weitere benötigte Softwarepakete unter `/usr/local/optub` installiert. Damit alle Programme und Bibliotheken gefunden werden, müssen einige Umgebungsvariablen gesetzt werden. Dies geschieht am einfachsten, indem man einmalig die Zeile `source /usr/local/optub/etc/satire_env.sh` in die Datei `.bash_profile` im eigenen Verzeichnis einfügt; bei jedem Login sollten danach alle Pfade automatisch richtig gesetzt sein.

Bei der Erstellung eines Analyseprogramms geht man wie folgt vor: Zunächst wird die Analysespezifikation in einer Datei mit der Endung `.optla` in einem neuen Verzeichnis mit demselben Namen abgelegt. Für die 'Live Variables'-Analyse würde man z. B. die Datei `live_variables/live_variables.optla` anlegen. Innerhalb des neuen Verzeichnisses führt man anschließend das Skript `newanalysis` aus. Dieses erzeugt ein Makefile sowie weitere Dateien, die für die Erstellung eines Analyseprogramms notwendig sind. Bei Änderungen an der Spezifikation ist ein erneuter Aufruf von `newanalysis` nicht nötig, es sei denn, die Zeile `carrier: ...` wurde geändert.

Mit dem Befehl `make` wird PAG aufgerufen, um aus der Spezifikation eine Analyseimplementierung zu generieren, und diese wird zu einem lauffähigen Programm übersetzt. Der Name des Programms entspricht dem Namen der Analysespezifikation, in obigem Beispiel also `live_variables`. Nach Änderungen an der Spezifikation erzeugt ein neuerlicher Aufruf von `make` eine aktualisierte Version des Analyseprogramms.

Das Analyseprogramm kann anschließend mit einem oder mehreren Beispielprogrammen als Argument aufgerufen werden. Dies führt die Analyse aus, produziert allerdings keine bleibenden Ergebnisse. Mit SATIrE erstellte Datenflussanalysen bieten verschiedene Möglichkeiten, Analyseergebnisse festzuhalten; die für unsere Zwecke wichtigste ist die Erstellung einer Visualisierung mit dem Kommandozeilenargument `--output-gdl=`*Dateiname*. Damit erstellt das Analyseprogramm eine visuelle Darstellung des Kontrollflussgraphen des Beispielprogramms, wobei an Programmpunkten zwischen den Anweisungen die jeweiligen Analyseinformationen dargestellt sind. Der Graph kann mit dem Programm `aisee` betrachtet werden.

Abbildung 1 zeigt ein kurzes Beispielprogramm und die GDL-Visualisierung des Ergebnisses, das die oben spezifizierte Analyse für dieses Programm berechnet. Jede Kante zwischen zwei Anweisungen ist mit der Analyseinformation in Form einer Menge von lebendigen Variablen versehen. Die Analyse gibt zur Laufzeit entsprechend der Transferfunktion für `DeclareStmt` eine Warnung über die nicht initialisierte Variable `n` aus.

```
void fac(void)
{
    int n, result;

    result = 1;
    while (n > 0)
    {
        result = result * n;
        n--;
    }
}
```

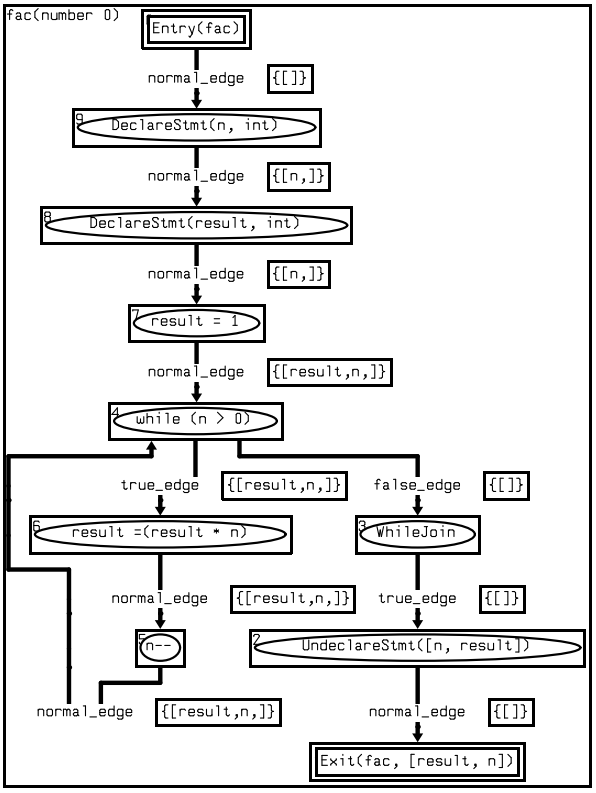

Abbildung 1: Beispielprogramm mit Ergebnis der 'Live Variables'-Analyse.

5 Weiterführende Dokumentation

Dieses Dokument kann nur einen ersten Eindruck der Entwicklung von Datenflussanalysen mit PAG und SATIrE geben. Näheres zu Syntax und Semantik der Spezifikationssprache FULA findet sich im Handbuch zu PAG, das auf dem Übungsrechner unter `/usr/local/optub/doc/PAG-Manual.ps` zu finden ist. Einen Überblick bietet Kapitel 3, die Kapitel 7 und 8 sind als Referenz zu verwenden. Kapitel 5 und 6 enthalten kurze Beispielspezifikationen, allerdings sind diese mit SATIrE nicht direkt verwendbar, da eine andere Programmzwischendarstellung verwendet wird.

Die Zwischendarstellung von SATIrE, d. h. die Menge aller möglichen Anweisungen, welche die Analyse antreffen kann, werden von der Baumgrammatik in der Datei `/usr/local/optub/doc/syn` beschrieben. Diese Grammatik beschreibt allerdings auch Sprachkonstrukte, die in der Übung nicht vorkommen werden; die einzelnen Übungsangaben geben genauere Informationen darüber, welche Teilsprache von C++ behandelt werden soll.

Die CFG-Attribute, Datentypen und Funktionen, die SATIrE für Datenflussanalysen zur Verfügung stellt, sind in `/usr/local/optub/doc/satire_attributes_auxiliary_functions` beschrieben. Eine weitere Quelle für (teils etwas veraltete) Informationen sind die SATIrE-FAQ, die im Web unter der URL `http://www.complang.tuwien.ac.at/satire/satirefaq` zu finden sind.