Preliminary Report

02242 Program Analysis E2010; Group 8

Andrei Lissovoi* Anton Makarov† Tomas Guomundsson‡

September 27, 2010

1 Overview

The eventual goal here is to construct a program that could, given a program in the Language of Guarded Commands (LoGC), be instructed to perform one or more of the transformations described in this report, and output the resulting program.

We plan on using **Java** to accomplish this; constructing a parser to create a syntax tree for the input program, implementing the three analyses as monotone framework instances drawing information from this syntax tree, and using the results of the analyses to perform transformations on the syntax tree. The transformed syntax tree could then be translated into an (optimized) program.

In broad terms, the parser output would be a tree-like structure of objects – where objects of different types would represent the various constructions and commands of the language; we could then label basic blocks and extract the flow function from this structure for use in the analyses. Statements could be removed from the tree to remove them from the program, or modified to accomplish things like constant propagation.

Some descriptions of the algorithms that follow may not fully reflect the idea that they will be implemented in terms of monotone frameworks.

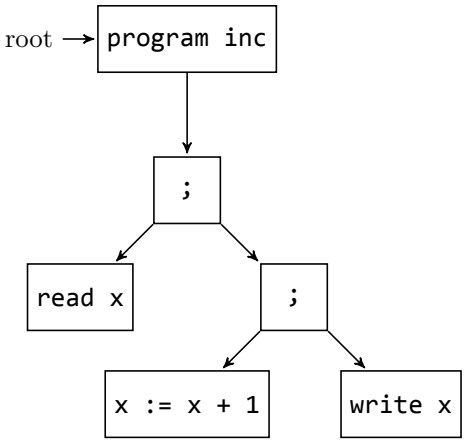

Figure 1: Syntax tree corresponding to `program inc: read x; x := x + 1; write x end`. Each node could be represented by an object; it is likely that expressions like `x + 1` are actually subtrees.

*s072442@student.dtu.dk
†s101565@student.dtu.dk
‡s102110@student.dtu.dk

2 Program Slicing

Given a program and a label, we want to remove any code that does not affect the behavior of the program at the labelled command.

1. Extract the flow function from the code.
2. Compute the reaching definitions function, using:

$$RD_{exit}(l) = (RD_{entry}(l) \setminus \{(x, a)\} \cup \{(x, l)\}) \qquad \text{for } \texttt{read} \quad x \text{ , and } x := \ldots$$
$$RD_{exit}(l) = RD_{entry}(l) \qquad \text{otherwise}$$

3. The assignment at the point of interest is in the program slice.
4. For every assignment statement $[x = e]^l$ in the program slice:
   (a) If a variable $v \in Var$ occurs in e, and $(v, l') \in RD_{entry}(l)$, then l' is in the program slice.
   (b) If the statement is within a guarded command, every guard condition in that guarded block is part of the program slice.

2.1 Program Examples

```
read x;
read z;
if x > 0 ->
  skip
[] true ->
  z := 5
fi;
z := z + 1
```

```
z = 42;
do true ->
  z = z + 1;
  do true ->
    skip
  od
od;
z := z + 1
```

Table 1: All commands in those two example programs are part of the program slice for the last line.

```
pi := 3.14;
read r;
d := 2 * r ;
x := d *  pi;
y := pi * r * r:
```

```
pi := 3.14;
read r;
y := 2 * r * r;
```

Table 2: A small program that calculates the area and circumference of a circle. So the program slice, if we focus only on the area, would be like so.

3 Dead Code Elimination

- Assignments to the variables that are never read again are *dead*.
- `skip` is always dead (but not always removable – syntactic requirements).
- Cannot reason about guard conditions, so cannot remove them.
- Cannot remove `read` statements – we assume the program receives input for those in a specific order.
- All variables are live at the program's exit.
  Or: all variables are dead, use `write e` to make variables in e live.
  Or: last statement is never dead code.

1. Compute the live variables function, using:

$$\begin{aligned} kill_{LV}([\texttt{x := e}]^l) &= \{x\} & gen_{LV}([\texttt{x := e}]^l) &= FV(e) \\ kill_{LV}([\texttt{skip}]^l) &= \emptyset & gen_{LV}([\texttt{skip}]^l) &= \emptyset \\ kill_{LV}([\texttt{abort}]^l) &= Var & gen_{LV}([\texttt{abort}]^l) &= \emptyset \\ kill_{LV}([\texttt{read x}]^l) &= \{x\} & gen_{LV}([\texttt{read x}]^l) &= \emptyset \\ kill_{LV}([\texttt{write e}]^l) &= \emptyset & gen_{LV}([\texttt{write e}]^l) &= FV(e) \\ kill_{LV}([\texttt{b -> }]^l) &= \emptyset & gen_{LV}([\texttt{b -> }]^l) &= FV(b) \end{aligned}$$

2. Every $[\texttt{x := e}]^l$ statement for which $x \notin LV_{exit}$ is dead and can be removed/replaced with `skip`.
3. Every statement of the form `skip; skip` can be replaced with `skip`.

3.1 Faint variables

It is possible to modify the definitions of LV to avoid more "dead" code by tracking faint variables: variables that are dead or only used to assign to other faint variables.

3.2 Program Examples

```
read x;
read y; # dead if read has no side-effects
read z;
y := 1; # dead always
y := 2; # dead if LV_exit = empty
a := z + x
```

Table 3: Live variables analysis performed on a simple program. Statements marked dead can be eliminated.

```
skip;
read x;
y := x;
z := 1;
do x > 1 ->
  z := z * x;
  x := x - 1;
od
x := 0;
write z
```

```
read x;
z := 1;
do x > 1 ->
  z := z * x;
  x := x - 1;
od
write z
```

Table 4: Dead code elimination performed on an example program; we assume that no variables are alive at the end.

4 Constant Folding

- If a variable is guaranteed to contain a constant when it is evaluated in an expression, we can replace it with the constant.

1. Compute the reaching definition function for the program.
2. Let C be a relation $\mathbf{Var}_\star \cup \{\texttt{?}\} \to Z \cup \{\texttt{?}\}$. Assume $C(l) = \texttt{?}$ until determined otherwise.
3. Examine each statement of the form `[x := e]`^{l} where $C(l) = \texttt{?}$. If e does not contain any variables, and therefore evaluates to a constant c, let $C(l) = c$.
   If no additional expressions have been discovered to evaluate to constants, this transformation is finished.
4. Examine each statement of the form `[x := e]`^{l}, `[e]`^{l} `->`, or `[write e]`^{l}, focusing on every variable z in e:
   - Let $L = \{l' \mid (z, l') \in RD_{entry}(l)\}$, the set of labels at which the variable could've been assigned prior to the evaluation of e.
   - If $\forall l', l'' \in L : C(l') = C(l'')$ and $\forall l' \in L : C(l') \neq \texttt{?}$, the variable z can be replaced by $C(l')$ in e.
5. Repeat steps 3-4.

General feeling: performance is largely determined by the order of steps 3-5 are performed in; doing something like a worklist algorithm here could result in acceptable run times.

4.1 Using Constant Propagation

In the spirit of not being lazy, constant propagation can also be used:

1. Perform the CP analysis, which computes which values a variable may assume at each label.
   The only new, interesting construction in our language is `read x`, which should mark x as $\top$.
2. Then, in every expression, replace any variables for which constant propagation produces a number at the label with that number.
   Simplifying expressions by performing arithmetic operations could be useful.

4.2 Program Examples

```
x := 6;
read w;
y := 4;
if x > w -> # x is 6
  y := 4
[] x <= w -> # x is 6
  y := 4
fi;
z := y * y; # y is 4 at all RDs
write z # z is 16
```

```
x := 6;
read w;
y := 4;
if 6 > w ->
  y := 4
[] 6 <= w ->
  y := 4
fi;
z := 16;
write 16
```

Table 5: Constant propagation performed on an example program; running live variable analysis afterward would eliminate the unnecessary assignments.