Preliminary report for Program Analysis 02242

Date: 2010-09-28

Authors:

s072425, Niels Thykier, s072425@student.dtu.dk

s072435, Melvin Winstrøm-Møller, s072435@student.dtu.dk

s072440, Morten Sørensen, s072440@student.dtu.dk

Programming language used: Java

Table of Contents

General definitions

When arguing for the correctness of algorithms, defining the flow and labels of the language is valuable. Below different definitions and functions is given, mirroring those of chapter 2 of the book. These include init(C), final(C), blocks(C), labels(C) and flow(C). Note that C is used instead of S to indicate commands instead of statements.

Labelling:

init(C):

init([x := a]_l) = l
init([skip]_l) = l
init([abort]_l) = l
init([read x]_l) = l
init([write x]_l) = l
init(C_1; C_2) = init(C_1)
init({C}) = init(C)
init(if gC fi) = init(gC)
init(do gC od) = init(gC)

init([e]_l -> C) = init(l)
init(gC1 [] gC2) = init(gC1)

final(C):

final([x := a]_l) = l
final([skip]_l) = l
final([abort]_l) = l
final([read x]_l) = l
final([write x]_l) = l
final(C_1; C_2) = final(C_2)

final({C}) = final(C)
final(if gC fi) = final_if(gC)
final(do gC od) = final_do(gC)

final_if([e]_l -> C) = final(C)
final_if(gC1 [] gC2) = final_if(gC1) union final_if(gC2)
(Since flow can only exit from one of the commands in ifs, never from the expressions).

final_do([e]_l -> C) = {[e]_l}
final_do(gC1 [] gC2) = final_do(gC1) union final_do(gC2)
(Since flow can only exit from one of the expressions in dos, never from the commands).

blocks(C):

blocks([x := a]_l) = {[x := a]_l}
blocks([skip]_l) = {[skip]_l}
blocks([abort]_l) = {[abort]_l}
blocks([read x]_l) = {[read x]_l}
blocks([write x]_l) = {[write x]_l}
blocks(C_1; C_2) = blocks(C_1) join blocks(C_2)
blocks({C}) = blocks(C)
blocks(if gC fi) = blocks(gC)
blocks(do gC od) = join blocks(gC)

blocks([e]_l -> C) = {[e]_l} join blocks(C)
blocks(gC_1 [] gC_2) = blocks(gC_1) join blocks(gC_2)

flow(C):

flow([x := a]_l) = Ø
flow([skip]_l) = Ø
flow([abort]_l) = Ø
flow([read x]_l) = Ø
flow([write x]_l) = Ø
flow(C_1; C_2) = flow(C_1) union flow(C_2) union { (l, init(C_2)) | l ∈ final(C_1) }
flow({C}) = flow(C)
flow(if gC fi) = flow(gC) union { ((l, l') | l, l' ∈ label(expressions(gC) and l != l') }
(An expression flows to its command,
and every expression flows to every expression).
flow(do gC od) = flow(gC) union
{ ((l, l') | l, l' ∈ label(expressions(gC) and l != l') } union
{ ((l, l') | l ∈ final(gC) and l' ∈ label(expressions(gC)) }
(An expression flows to its command,
and every expression flows to every expression,
and every final labels of commands goes to every expression).

flow([e]_l -> C) = {(l, init(C))} union flow(C)
flow(gC1 [] gC2) = flow(gC1) union flow(gC2)

expressions([e]_l -> C) = {[e]_l}

expressions(gC1 [] gC2) = expressions(gC1) union expressions(gC2)

used(C):

used([x := a]) = fv(a)
used([skip]) = Ø
used([abort]) = Ø
used([read x]) = Ø
used([write x]) = {x}
used([e]) = fv(e)

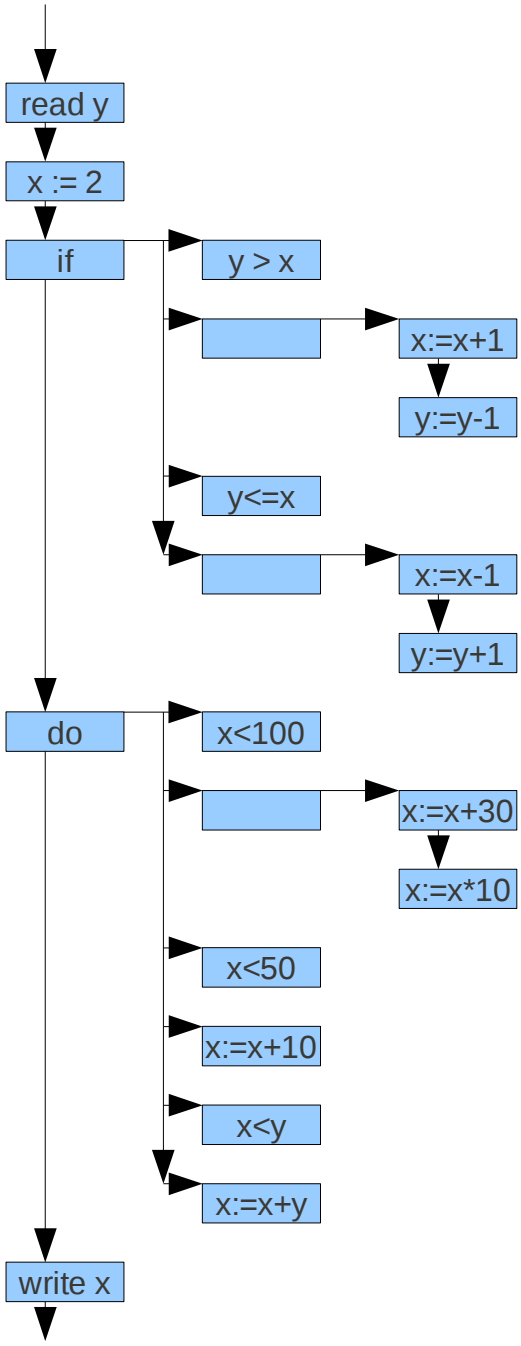
Data structure

For structuring the program in memory, we have decided to build it in an abstract syntax tree-like structure. While parsing the program, we are applying a simplification, all statements are using at most three addresses in memory. This makes it easier to work with and easier to see what is happening. When parsing the program, every statement get its own node in this tree. There are a few exceptions however, for how to handle if- and do-statements? The if- or do-statement itself get its own block, leading to whatever the statement contains. The structure in the branch for both if- and do-statements is the same. Every second block, starting from the first one, is a condition block. Every second block, starting from the second block, is the action-block. In case block five is true, then block six is the one which should be executed next. In case any of the action or condition blocks contains more than one statement, it creates a new branch which contains all the statements.

As an example, we can use the following program:

```
read y;
x:=2;
if y > x -> x := x+1; y := y-1;
[] y <= x -> x := x-1; y := y+1;
fi;
do x < 100 -> x := x+30; x := x*10;
[] x < 50 -> x := x+10;
[] x < y -> x := x+y;
od;
write x;
```

With that, we can now construct the tree:



Since this will be implemented in Java, there are some design choises. One of these is, how should the data be stored? The solution for this question will have a huge impact on running time and memory consumption. There are no perfect solution, it will be a tradeoff between those two parameters. Choosing a solution which have a high running time for search and/or add, will have a very large impact when analysing an entire program. For our solution, we will need some kind of list which allows random access. A Vector or ArrayList, or even a simple array of nodes, could be used. Using a queue or stack will make it easy to add or remove the top or bottom element (depending of the queue/stack used), however searching through the queue/stack can take a large amount of time. Another choise which needs to be decided is how to refer to a block. Adding a label to each block, and then storing them all in some large array, could be used and is not a bad idea. However, as far as we can see, we can use references to blocks instead of using labels. In many cases, we are not particular interested in exactly where the block is. However in the case with program slicing, we are interested if the block is part of an if- or do-statement. For solving that, every node in our tree will have a reference to its parent. With that, we can trace it to an if/do-parent. For nodes in the main branch of the tree, they will have an imaginary root node as its parent.

An example of the Node class could be:

```
class Node {

	private Node parent;
	private int childNo;
	...

	public Node(Node parent, int childNo, ...) {
		this.parent = parent;
		this.childNo = childNo;
		...
	}

	...
)
```

Reaching Definitions

The implementation of Reaching Definitions (RD) will be based on the work-list algorithm for solving the equations for the entry part of RD, as presented in the course lectures. Although the implementation of the algorithm itself will be the same as that of the slides, the extension of the While-language requires new definitions for the functions used in the algorithm: kill_RD(l) and gen_RD(l).

kill_RD(l):

kill_RD([x := a]_l) = {(x, ?)} join {(x, l') | B_l' is an assignment to x in C*}
kill_RD([read x]_l) = {(x, ?)} join {(x, l') | B_l' is an assignment to x in C*}
(Same as assigning to x).
kill_RD([skip]_l) = Ø
kill_RD([abort]_l) = {(x, l) | l ∈ labels(C*) and x ∈ FV(C*) }
(If we abort, no variables reach any further. So, upon an abort, all is killed. Note that abort should not flow anywhere).
kill_RD([write x]_l) = Ø
(Same as reading x).
kill_RD([if gC fi]_l) = Ø
kill_RD([do gC od]_l) = Ø
(For both if and do, the start labels do nothing, neither reading or writing. They are pure flow elements).
kill_RD([e]_l) = Ø
(Nothing is written in a test).

gen_RD(l):

gen_RD([x := a]_l) = {(x, l)}
gen_RD([read x]_l) = {(x, l)}
(Same as assigning to x).
gen_RD([skip]_l) = Ø
gen_RD([abort]_l) = Ø
gen_RD([write x]_l) = Ø
(Same as reading x).
gen_RD([if gC fi]_l) = Ø
gen_RD([do gC od]_l) = Ø
(For both if and do, the start labels do nothing, neither reading or writing. They are pure flow elements).
gen_RD([e]_l) = Ø
(Nothing is written in a test).

Program Slicing

Program slicing is defined as:
“Given a program the idea is to determine the part of the program that may influence the values computed at a given point of interest; this part of the program is then called a program slice. ”

A simple example is:

1: a := 3;
2: b := 5;
3: x := a;
[Point of interest; 3. Var: x.]

Which is turned into:

1: a := 3;
3: x := a;

Control flow

An important issue regarding program slicing is that not only assignment, but also program control flow is important. For instance, consider the following example:

```
1: a := 3;
2: b := 4;
3: if     (b = 4) -> a := 5; []
4:        (b = 3) -> a := 10000;
5: fi;
6: x := a;
[Point of interest; 6. Var: x.]
```

If control flow is ignored, the program slice might end up looking like this:

```
4: a := 10000;
6: x := a;
```

which is clearly wrong. Instead, the correct program slice must be:

```
2: b := 4;
3: if     (b = 4) -> a := 5; []
4:        (b = 3) -> a := 10000;
5: fi;
6: x := a;
```

Note that b is included, since it helps determine the flow: it is included in at least one of the conditions in the guarded if. Furthermore, the first assignment to a is gone, since at least one of the statements in line 3 and 4 is executed; else, the program would have aborted before line 6.
Based on these considerations, control flow must be included, at least to some degree.

To consider control flow properly, both the guarded if and the guarded do will be considered. Early termination and non-termination is also considered.

Guarded do:

For the guarded do, just the conditions with commands that includes definitions which reaches referenced variables are included in the program slice.. This claim is justified below:

Consider some program:

```
1: a := 42;
2: do    (P) -> a := 30; S1;
3:       (Q) -> S2;
4: od
5: x := a;
```

Assume that S2 does not contain any assignments to a. It is clear that the predicate P is relevant to a, and therefore to x. It should therefore be included in the program slice. Instead, the relevance of Q -> S2 is not determined yet. For S2, there are two cases: If it causes definitions that reaches P, it is relevant, and would be included in a use-definition run on the variables of P at P. If it instead has no

definitions that reaches P, it is irrelevant to the values of a. No matter if it is true or false, the do will still be executed as long as P is true, and since S1 does not touch P, it has no effect how many times a := 30; S1; is executed. Therefore, the correct program slice is:

```
1: a := 42;
2: do    (P) -> a := 30; S1';
4: od
5: x := a;
```

Note that only the parts of S1 affecting P or a should be included.

Based on this, for guarded do's, just the conditions with commands that includes definitions which reaches referenced variables are included in the program slice.

Guarded if:

For the guarded if, all the conditions should be included in the program slice if any definition in any of their commands reaches the referenced variables. This claim is justified below:

Consider 2 different programs:

```
1: a := 2;
2: if     (P) -> a := 3; []
3:        (Q) -> a := 4; []
4:        (R) -> z := 0;
5: fi;
6: x := a;
```

Here, it may seem clear that line 4 is irrelevant. But it is in fact still relevant. Assume that line 4 is removed in the program slice. In this case, the definition from line 1 will never reach line 6, since it will either be overwritten in line 2 and 3, or the program will terminate before line 6. But this is different from the semantics of the original program; here, the definition from line 1 does reach line 6, since line 4 does not assign to a.

Based on this, for guarded if's, all the conditions should be included in the program slice if any definition in any of their commands reaches the referenced variables.

Non-termination and early termination:

Non-termination and early termination may have an effect on whether or not the point of interest is ever reached, but not directly which values that may be computed. Therefore, non-termination and early termination is ignored. For instance, look at the following program:

```
1: a := 4;
2: do    (true) -> skip;
3: od;
4: x := a;
```

The correct program slice would be:

```
1: a := 4;
4: x := a;
```

Note that this is also a very practical definition, partly because non-termination is generally undecidable.

Corner cases

A corner case of the program is that a condition may be included without its command is included. This is the case for the guarded if, since all conditions are included whether or not there is something relevant in their commands. For instance,

```
1: a := 2;
2: if     (P) -> a := 3; []
3:        (R) -> z := 0;
4: fi;
5: x := a;
```

would end up looking like:

```
1: a := 2;
2: if     (P) -> a := 3; []
3:        (R) ->
4: fi;
5: x := a;
```

This is not legal semantics. In these cases, the algorithm should repair this “damaged” guarded statement by providing a skip statement, like this:

```
1: a := 2;
2: if     (P) -> a := 3; []
3:        (R) -> skip;
4: fi;
5: x := a;
```

Algorithm

Now, an algorithm is constructed. The basis of the algorithm is to have a Worklist W of which block to investigate next, a set of blocks PS that represents the Program Slice and a block POI (Point Of Interest).

```
//Setup everything, and keep consuming W as long as it is not empty.
1: Initialize W and PI to the empty set, and calculating Reaching Definitions on the program.
2: If POI assigns (ie. if POI is an assignment or a read command), add the POI to PS.
3: Add POI to W.
4: While W is not empty, do:
        Consume(head(W), W, PS)
        W = tail(W)
5: Repair all “damaged” guarded commands in PS of the form (e -> ) by turning them into (e -> skip).

//Consume does 2 things:
//      It calculates RD to determine the data flow, and adds relevant data flow blocks.
//      It finds the relevant guarded ifs and dos to determine the control flow, and adds relevant
//      control flow blocks.
Consume(B block, Worklist W, Set<Block> PS) {
1: For the block, run “used” on it to get the interesting variables.
2: For each variable found, use the analysis of RD to find all the relevant labels, and Add them.
3: C = B
4: While C has a parent P, find P and do the following:
        If P is a command of the form (do gC od), there are two cases for C:
                C is an expression: Add(C).
                C is a command: Add the corresponding expression.
        Else, if P is a command of the form (if gC fi):
                Add all expressions in child(P).
        Else, C = P and continue.

Add(Block b) {
1: If B not part of PS, add to PS and W.
}
```

The children is defined as:

```
child(do gC od)         = expansion(gC)
child(if gC fi)         = expansion(gC)
child(C1; C2)           = {C1, C2}
child({C})              = C
child(_)                = Ø

expansion(e -> C)               = {e, C}
expansion(gC1 [] gC2)           = expansion(gC1) union expansion(gC2)
```

A P is a parent of C iff C belongs to the set of child(P).

Characteristics:

N = number of blocks in the program.
The while in line 4 will have at most N iterations, since no block will be added more than once to the worklist, and one work-block is always consumed on each iteration of the while. Consume will therefore be run at most N times.
For Consume, the while in line 4 takes at most N time, since each block will be handled in line 4 at most one time (either as a parent or as a guarded command/expression), and there is at most N blocks.
Since all other operations are at most polynomial, the algorithm is therefore polynomial.

Dead code elimination

Before we can eliminate dead code, we have to first find out which variables are used where. For that, we can start with performing the Live Variable analysis.

Live variable Analysis

Given our monotone frameworks and the variables for the Live Variable analysis, we get the following data flow equations:

LV_exit(l) >=	if (l ∈ final(S_*)) then (Ø)
	else (U {LV_entry(l') | (l',l) ∈ flow^R(S_*)})
LV_entry(l) >=	(LV_exit(l)\kill_LV(B^l)) U gen_LV(B^l)
	where B^l ∈ blocks(S_*)

From the data flow equations, we can see that it is a may-analysis (we use union and not intersection), and it is performed backwords (flow^R(S_*) is used). Before we continue, there is something that should be noted. In order to create an even smaller solution to our "problem", in gen_LV(l) we are only generating anything if the variable assigned is in our list of live variables. As an example, we can use the following code:

```
read x;
y := 5;
if (x > y) -> x := 1;
[] (x <= y) -> x := 2;
fi;
z := 2;
write x;
```

The labels will be distributed as follows:

[read x](1)

[y := 5](2)

if [x > y](3) -> [x := 1](4)

[] [x <= y](5) -> [x := 2](6)

fi;

[z := 2](7)

[write x](8)

In our example code, we can see two different statements for which kill_LV(l) and gen_LV(l) is not defined yet, namely [read x] and [write x]. For [read x] we can treat it as [x := a] where a is whatever value is read. Basicly it is an assign statement for the variable x, we just don't know what the value is. For [write x] the variable x is clearly used and gen_LV(l) should just blindly add the variable to our list; kill_LV(l) should not eliminate anything for [write x].

Following our data flow equations, we can now write a table over what is generated and killed in each block:

l | kill_LV(l) | gen_LV(l)

---

1 | {x} | Ø

2 | {y} | Ø

3 | Ø | {x,y}

4 | {x} | Ø

5 | Ø | {x,y}

6 | {x} | Ø

7 | {z} | Ø

8 | Ø | {x}

According to our extra rule to gen_LV(l) label 3 and 5 should not generate any variables, however since they have an effect on a variable which we find interesting (x) we now find both of them interesting and therefore adds them to our list.

We can then write up our equations:

LV_entry(1) = LV_exit(1)\{x}

LV_entry(2) = LV_exit(2)\{y}

LV_entry(3) = LV_exit(3) U {x,y}

LV_entry(4) = LV_exit(4)\{x}

LV_entry(5) = LV_exit(5) U {x,y}

LV_entry(6) = LV_exit(6)\{x}

LV_entry(7) = LV_exit(7)\{z}

LV_entry(8) = {x}

LV_exit(1) = LV_entry(2)

LV_exit(2) = LV_entry(3) U LV_entry(5)

LV_exit(3) = LV_entry(4)

LV_exit(4) = LV_entry(7)

LV_exit(5) = LV_entry(6)

LV_exit(6) = LV_entry(7)

LV_exit(7) = LV_enrty(8)

LV_exit(8) = Ø

Lastly, we will then get our smallest solution:

l	LV_entry(l)	LV_exit(l)
1	Ø	{x}
2	{x}	{x,y}
3	{x,y}	Ø
4	Ø	{x}
5	{x,y}	Ø
6	Ø	{x}
7	{x}	{x}
8	{x}	Ø

Use-Definition and Definition-Use chains

Now that everything about the Live Variable analysis is completed, we can now construct a du-(define-use) or ud-chain (use-define). For that we can write up the following definitions:

ud(x,l') = {l | def(x,l) and exists l'' : (l,l'') . flow(S_*) &and clear(x,l'',l')} U {? | clear(x,init(S_*),l')}

du(x,l) = if (l = ?) then ({l' | clear(x,init(S_*),l')}) else ({l' | def(x,l) &and &exists l'' : (l,l'') .

flow(S_*) &and clear(x,l'',l')})

A couple of predicates is used, namely clear(x,l,l'), use(x,l) and def(x,l). The predicate def(x,l) checks whether the variable is used in a block with the following definition:

def(x,l) = (exists B : [B]^l ∈ blocks(S_*) and x ∈ kill_LV([B]^l))

The second predicate, use(x,l), checks whether the variable is used in a block:

use(x,l) = (exists B : [B]^l ∈ blocks(S_*) and x ∈ gen_LV([B]^l))

The last predicate, clear(x,l,l'), checks whether there is a definition clear path from label l to l'. The meaning of a definition clear path, is to check whether there is a clear path (as in the variable is not redefined) between label l and l'.

clear(x,l,l') = exists l_1, ..., l_n : (l_1 = l) and (l_n = l') and (n > 0) and (&all i ∈ {1, ..., n-1} : (l_i,l_i+1) ∈ flow(S_*)) and (all i ∈ {1, ..., n-1} : not def(x,l_i)) and use(x,l_n)

With these defined, we can now construct our chains. We are interested to see where a variable is used after it have been defined, so a du-chain is needed:

du(x,l)	x	y	z
1	{3,5}	Ø	Ø
2	{3,5}	Ø	Ø
3	Ø	Ø	Ø
4	{8}	Ø	Ø
5	Ø	Ø	Ø
6	{8}	Ø	Ø
7	Ø	Ø	Ø
8	Ø	Ø	Ø

Now that the du-chain is written, we can check which lines of code we can remove. A way to do that, is to go through the du-chain and the table over what is killed in which labels. Every time a variable is killed, we can look up in the du table to see where the variable is used. For label 1, we can see that variable x is killed; in the du table, we can see that x is used at label 3 and 5, so that line will not be killed. Label 7 on the other hand, there is z killed. However in the du table, the variable z is not used anywhere. Therefore we can eliminate label 7 since it is not used anywhere. Our resulting code will then be this:

```
read x;
y := 5;
if (x > y) -> x := 1;
[] (x <= y) -> x := 2;
fi;
write x;
```

If this was the very first analysis to be executed on the code, this case is highly unlikely. Why should the programmer define variables which never is used? However this might be the case after some other analysis have been executed. If some else analysis have eliminated some useless code, it might not have cleaned up completely after itself and the variable z is left behind.

Constant folding

Constant folding is a must analysis where variables can only be substituted with constants only if the variables are set the same constant regardless of the path taken to l. The unhandled cases for the monotone framework are the following two statements:

[read x]: f(sigma) = sigma[x -> “top”]

[write x]: f(sigma) = sigma

Implementation wise it is possible to use a table, where a variable x is mapped to the tuple (t,v), where t is one of “undefined”, “unknown”, “constant” and v is an integer. The table works in the following way:

- If the variable x is not in the table, then the table lookup will return (“undefined”, ?).
- Analyse the current statement; if the statement assigns a value to a variable x (e.g. x:=a) then one of the following should be done, otherwise the statement is ignored.
    - If a can be evaluated to a constant value (or it is a constant value) then the analysation results in (“constant”, v), where v is the constant value of evaluating a. If a contains variables it may still be possible to evaluate a to a constant if enough variables are known to be constant.
    - If a cannot be evaluted to a constant (e.g. due to lack of information), then the analysation results in (“unknown”, ?). “read x” is an example of this case.
- Inserting in the table is done by looking up the variable in the table and acting based on the following cases:
    - If x is not already in the table, then insert (t,v) generated from the analysed statement.
    - If x is already in the table and the look up results in (“unknown”, ?), then then nothing is done.

- If x is already in the table and the look up results in (“constant”, v) and the analysation resulted in (“unknown”, ?), then replace the current mapping with (“unknown”, ?)
- If x is already in the table and the look up results in (“constant”, v1) and the analysation results in (“constant”, v2) then either insert (“unknown”, ?) if v1 != v2, otherwise do nothing.