02242 - Program Analysis

Final Report

s171783 - Onat Buyukakkus
s181314 - Altug Tosun

09.09.2018

Contents

1 Foundation

1.1 MICRO-C

Basic actions are atomic operations which means that in a flow graph basic actions are nodes and in a program graph basic actions are edges.
Basic actions in MICRO-C are:

- **Declerations:** $D ::= int\ x;\ |\ int[n]\ A;\ |\ \{int\ fst;\ int\ snd\}\ R;$
- **Statements:** $S ::= l := a;\ |\ R := (a_1, a_2);\ |\ read\ l;\ |\ write\ a;$
- **Expressions:** $b ::= true\ |\ false\ |\ a_1\ opr\ a_2\ |\ b_1\ opr\ b_2\ |\ not\ b$

1.2 Control Flow Analysis

1.2.1 Flow Graphs

Algorithm for generating flow graphs of MICRO-C programs is as follows where $flow(S)$ is the edges of the flow graphs for S. There is an edge from one node to another if the flow of control may go from the first node to the second; labels are introduced to distinguish between the actions.

$$
\begin{aligned}
flow([int\ x;]^l) &= \emptyset \\
flow([int[n]\ A;]^l) &= \emptyset \\
flow([\{int\ fst;\ int\ snd\}\ R;]^l) &= \emptyset \\
flow(D_1\ D_2) &= flow(D_1) \cup flow(D_2) \cup \{(l,\ init(D_2)) \mid l \in final(D_1)\} \\
flow([l := a;]^l) &= \emptyset \\
flow([R := (a_1, a_2);]^l) &= \emptyset \\
flow([read\ k;]^l) &= \emptyset \\
flow([write\ l;]^l) &= \emptyset \\
flow(S_1\ S_2) &= flow(S_1) \cup flow(S_2) \cup \{(l,\ init(S_2)) \mid l \in final(S_1)\} \\
flow(if\ [b]^l\ \{S_0\}) &= \{(l,\ init(S_0))\} \cup flow(S_0) \\
flow(if\ [b]^l\ \{S_1\}\ else\ \{S_2\}) &= flow(S_1) \cup flow(S_2) \cup \{(l,\ init(S_1)), (l,\ init(S_2))\} \\
flow(while\ [b]^l\ \{S_0\}) &= \{(l,\ init(S_0))\} \cup flow(S_0) \cup \{(l',\ l) \mid l' \in final(S_0)\}
\end{aligned}
$$

1.2.2 Program Graphs

Algorithm for generating program graphs of MICRO-C programs is as follows where $edge(S)$ is the edges of the program graph for S. The initial node is q_0 , the final node is $q_\bullet$ and the edge is labelled with the action taking place; new nodes are created on the fly.

$$
\begin{aligned}
edges(q_0 \rightsquigarrow q_\bullet)[\![int\ x;]\!] &= \{(q_o,\ int\ x,\ q_\bullet)\} \\
edges(q_0 \rightsquigarrow q_\bullet)[\![int[n]\ A;]\!] &= \{(q_o,\ int[n]\ A,\ q_\bullet)\} \\
edges(q_0 \rightsquigarrow q_\bullet)[\![\{int\ fst;\ int\ snd\}\ R;]\!] &= \{(q_o,\ \{int\ fst;\ int\ snd\}\ R,\ q_\bullet)\} \\
edges(q_0 \rightsquigarrow q_\bullet)[\![D_1\ D_2]\!] &= edges(q_0 \rightsquigarrow q)[\![D_1]\!] \cup edges(q \rightsquigarrow q_\bullet)[\![D_2]\!]
\end{aligned}
$$

$$
\begin{aligned}
&\textit{where } q \textit{ is } \textit{fresh}\\
\textit{edges}(q_0 \rightsquigarrow q_\bullet)[\![l := a;]\!] &= \{(q_o,\ l := a,\ q_\bullet)\}\\
\textit{edges}(q_0 \rightsquigarrow q_\bullet)[\![R := (a_1,\ a_2);]\!] &= \{(q_o,\ R := (a_1,\ a_2),\ q_\bullet)\}\\
\textit{edges}(q_0 \rightsquigarrow q_\bullet)[\![\textit{read } k;]\!] &= \{(q_o,\ \textit{read } k,\ q_\bullet)\}\\
\textit{edges}(q_0 \rightsquigarrow q_\bullet)[\![\textit{write } l;]\!] &= \{(q_o,\ \textit{write } l,\ q_\bullet)\}\\
\textit{edges}(q_0 \rightsquigarrow q_\bullet)[\![S_1\ S_2]\!] &= \textit{edges}(q_0 \rightsquigarrow q)[\![S_1]\!] \cup \textit{edges}(q \rightsquigarrow q_\bullet)[\![S_2]\!]\\
&\textit{where } q \textit{ is } \textit{fresh}\\
\textit{edges}(q_0 \rightsquigarrow q_\bullet)[\![\textit{if } (b)\ \{S_0\}]\!] &= \{(q_0,\ b,\ q),\ (q_0,\ \neg b,\ q_\bullet)\} \cup \textit{edges}(q \rightsquigarrow q_\bullet)[\![S_0]\!]\\
&\textit{where } q \textit{ is } \textit{fresh}\\
\textit{edges}(q_0 \rightsquigarrow q_\bullet)[\![\textit{if } (b)\ \{S_1\}\ \textit{else}\ \{S_2\}]\!] &= \{(q_0,\ b,\ q_1),\ (q_0,\ \neg b,\ q_2)\} \cup \textit{edges}(q_1 \rightsquigarrow q_\bullet)[\![S_1]\!]\\
&\quad \cup \textit{edges}(q_2 \rightsquigarrow q_\bullet)[\![S_2]\!]\\
&\textit{where } q1,\ q2 \textit{ are } \textit{fresh}\\
\textit{edges}(q_0 \rightsquigarrow q_\bullet)[\![\textit{while } (b)\ \{S_0\}]\!] &= \{(q_0,\ b,\ q),\ (q_0,\ \neg b,\ q_\bullet)\} \cup \textit{edges}(q \rightsquigarrow q_0)[\![S_0]\!]\\
&\textit{where } q \textit{ is } \textit{fresh}
\end{aligned}
$$

1.2.3 $init(S)$ and $final(S)$ Functions

$init(S)$ and $final(S)$ functions for MICRO-C is shown in Table 1.

S	init(S)	final(S)
$[int\ x;]^l$	l	$\{l\}$
$[int[n]\ A;]^l$	l	$\{l\}$
$[\{int\ fst;\ int\ snd\}\ R;]^l$	l	$\{l\}$
$D_1\ D_2$	$init(D_1)$	$final(D_2)$
$[l := a;]^l$	l	$\{l\}$
$[R := (a_1, a_2);]^l$	l	$\{l\}$
$[read\ k;]^l$	l	$\{l\}$
$[write\ l;]^l$	l	$\{l\}$
$S_1\ S_2$	$init(S_1)$	$final(S_2)$
$if\ [b]^l\ \{S_0\}$	l	$\{l\} \cup final(S_0)$
$if\ [b]^l\ \{S_1\}\ else\ \{S_2\}$	l	$final(S_1) \cup final(S_2)$
$while\ [b]^l\ \{S_0\}$	l	$\{l\}$

Table 1: $init(S)$ and $final(S)$ functions for MICRO-C

1.2.4 Example Program

This section considers following example MICRO-C program and uses above algorithms to generate both flow and program graphs.

```
1  int[3] a;              <l1>
2  int x;                 <l2>
3  x := 0;                <l3>
4  while x<3 {            <l4>
5      a[x] := 1;         <l5>
6      if x==2 {          <l6>
7          a[x] := 3;     <l7>
8      }
9  }
10 write a[0]+a[2];       <l8>
```

Program graph for above example program is drawn in Figure 1.

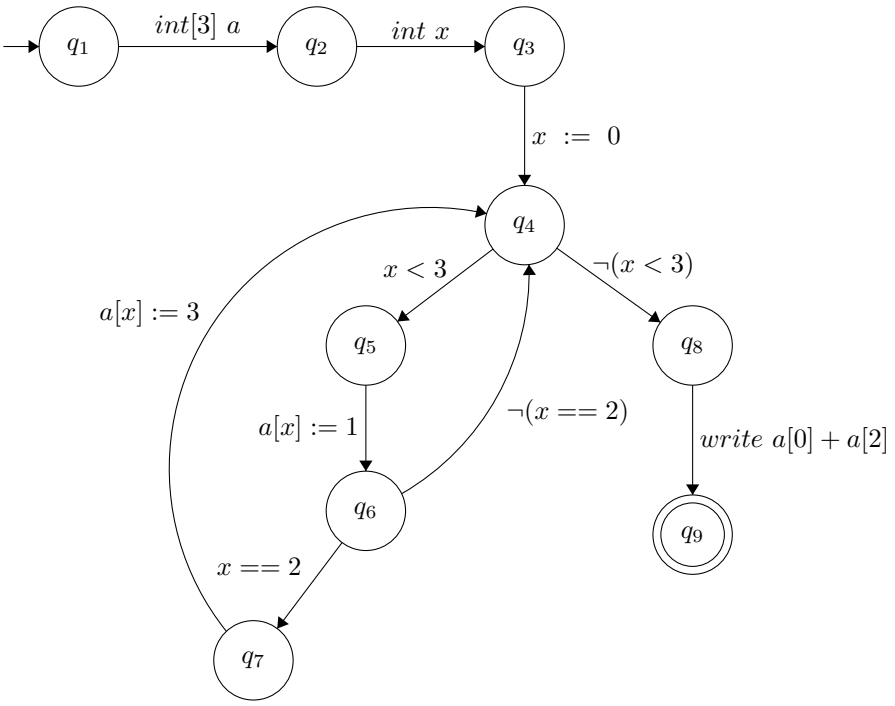


Figure 1: Program graph for example MICRO-C program

Flow graph for above example program is drawn in Figure 2.

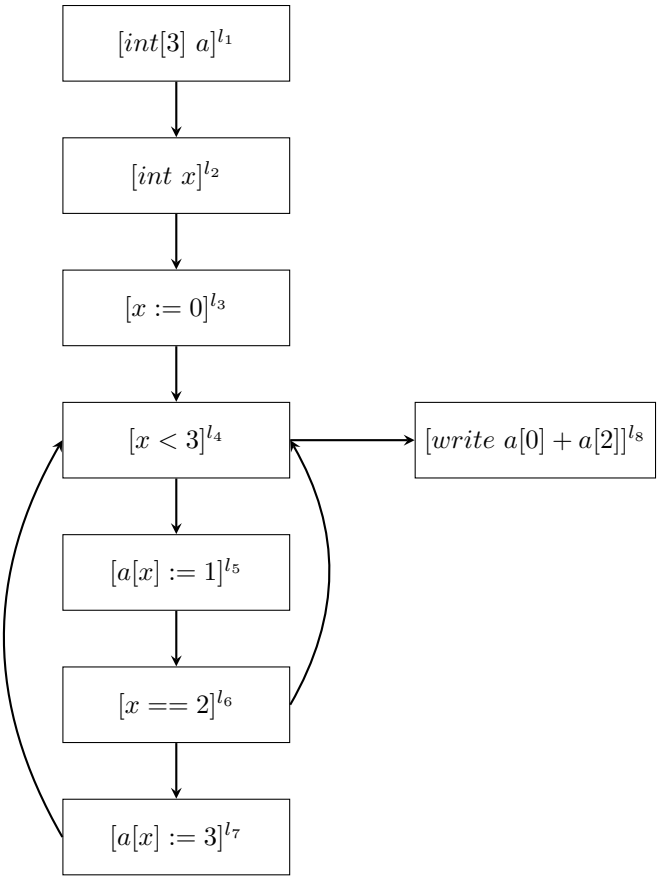


Figure 2: Flow graph for example MICRO-C program

1.2.5 Extensions

We have decided to add $[break]^l$ and $[continue]^l$ program steps as two extensions. We will demonstrate an example:

```
1 while a>b {                          <l1>
2     x := a+1;                        <l2>
3     if x==5 {                        <l3>
4         break; (or continue;)        <l4>
5     }
6     x = x +1                         <l5>
7 }
8 a := 1;                              <l6>
```

Flow graphs

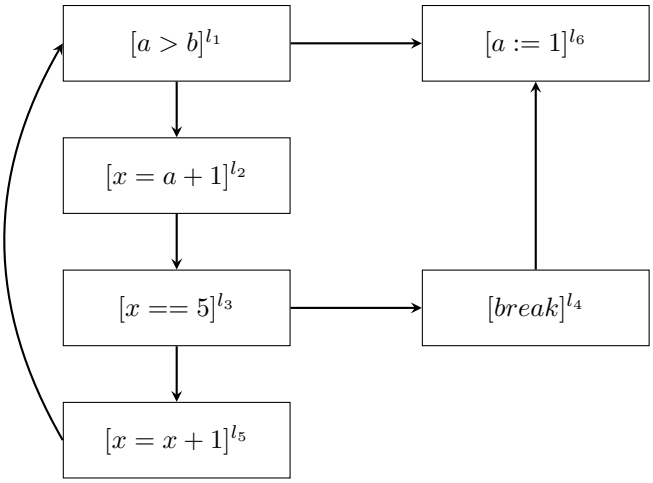


Figure 3: Flow graph example with a break

While generating a flow graph, if we see a *break* statement, the *break* node should not be connected to next program statement. In the example above, one can see that l_4 and l_5 are not connected. In order to achieve that, $final([break]^l)$ should be equal to $\emptyset$.

In order to connect l^4 to l^6 in the example, we need to keep track of which node should the *break* statement connect to after it breaks the loop. We can modify $flow$ function to take this information as a parameter. Then $flow([break]^l, l') = (l, l')$ where l' is the information that we keep track of at the beginning of the loop.

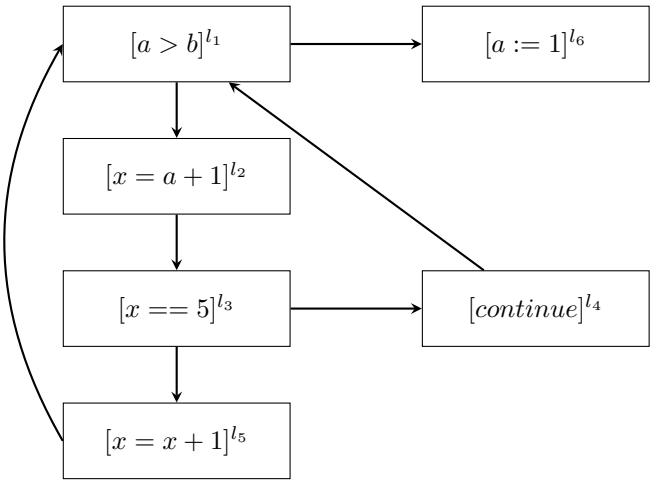


Figure 4: Flow graph example with a continue

For continue statements, again in order not to connect l^4 to l^5 in the above example, $final([continue]^l)$ should be equal to $\emptyset$.

Also, in order to connect l^4 to l^1 in the example, we need to keep track of which node should the *continue* statement connect to after it continues through the next iteration. We can

modify $flow$ function to take this information as a parameter. Then $flow([continue]^l, l') = (l, l')$ where l' is the information that we keep track of at the beginning of the loop.

This solution arises a point to think about: nested loops. We need to update the information as we go through the inner loops every time so that every loop has its own information about where the *break* or *continue* should connect to.

Program graphs

Same solution also works for the program graphs, again we need to keep track of the node where the *break* or *continue* statement will connect to. So $edges(q_0 \rightsquigarrow q_\bullet, q_l)[\![break;\ (or\ continue)]\!]$ is equal to $\{q_0, break\ (or\ continue), q_l\}$ where q_l is the information that we keep track of at the beginning of the loop.

1.2.6 Notes

This report uses the flow graph representation for the theoretical part and program graph representation for the the implementation part because we wanted to work with both of them. This is solely a preference since two graph representations are equally good and interchangeable.

2 Analyses

2.1 Reaching Definitions Analysis

Reaching Definitions Analysis determines for each program point, which assignments may have been made and not overwritten, when program execution reaches this point along some path.

Approach is first constructing the $kill_{RD}$ and gen_{RD} sets of reaching definitions for all the actions of the program. Next, constructing a set of equations defining the reaching definitions at the various nodes of the flow graph and finally using an algorithm to solve the equations.

RD_0 and $RD_\bullet$ equations used in this analysis are below.

$$RD_\bullet = (RD_0 \backslash kill_{RD}(B^l)) \cup gen_{RD}(B^l)$$

$$RD_0 = \begin{cases} \{(x,?)|x \in FV(S_*)\} & if\ l = init(S_*) \\ \cup\{RD_\bullet(l')|(l',l) \in flow(S_*)\} & otherwise \end{cases}$$

2.1.1 Handling Arrays and Records

Addition to WHILE language, MICRO-C offers arrays and records as data types. Way these two data types are handled plays an important role for this analysis since it changes $kill_{RD}$ and gen_{RD} sets thus the equations.

One can select between two ways of handling arrays and records for this analysis.

First option is to amalgamate the components which means acting as if arrays and records were a single variable like $int\ x$. In this option one does not need to keep track of assignments to elements of arrays and records individually. Instead, whole array and record is treated as changed when there is an assignment to an individual element.

Second option is to deal every element of arrays and records individually.

For arrays, first option is way more suitable because generally array elements are reached with a variable like $A[x] := 3$. If we were using second option for arrays, since we cannot determine x before execution, when we encounter such assignment we would need to say that all elements of the array might have been changed. Problem with this approach is memory: storing every element of the array in every modification is not memory friendly at all. Also, there would be no point of this approach since we do not gain anything.

However for records, elements are reached with keywords fst and snd so we always know which element has modified. For getting more information on our analysis second option is more suitable for records since there is no memory issues.

2.1.2 $kill_{RD}$ and gen_{RD} Sets For MICRO-C

$kill_{RD}$ and gen_{RD} sets for MICRO-C are as follows.

$$\begin{aligned} kill_{RD}([int\ x]^l) &= \{(x,?)\} \\ kill_{RD}([int[n]\ A]^l) &= \{(A,?)\} \\ kill_{RD}([\{int\ fst;\ int\ snd\}\ R]^l) &= \{(R.fst,?),(R.snd,?)\} \end{aligned}$$

$$
\begin{aligned}
kill_{RD}([x := a]^l) &= \{(x, ?)\} \cup \{(x, l') | B^{l'} \textit{ is an assignment to } x\} \\
kill_{RD}([A[a] := a']^l) &= \{(A, ?)\} \cup \{(A, l') | B^{l'} \textit{ is an assignment to an element of } A\} \\
kill_{RD}([R.fst := a]^l) &= \{(R.fst, ?)\} \cup \{(R.fst, l') | B^{l'} \textit{ is an assignment to } R.fst\} \\
kill_{RD}([R.snd := a]^l) &= \{(R.snd, ?)\} \cup \{(R.snd, l') | B^{l'} \textit{ is an assignment to } R.snd\} \\
kill_{RD}([R := (a_1, a_2)]^l) &= \{(R.fst, ?)\} \cup \{(R.snd, ?)\} \\
&\quad \cup \{(R.fst, l') | B^{l'} \textit{ is an assignment to } R.fst\} \\
&\quad \cup \{(R.snd, l') | B^{l'} \textit{ is an assignment to } R.snd\} \\
kill_{RD}([read\ x]^l) &= \{(x, ?)\} \cup \{(x, l') | B^{l'} \textit{ is an assignment to } x\} \\
kill_{RD}([read\ A[a]]^l) &= \{(A, ?)\} \cup \{(A, l') | B^{l'} \textit{ is an assignment to an element of } A\} \\
kill_{RD}([read\ R.fst]^l) &= \{(R.fst, ?)\} \cup \{(R.fst, l') | B^{l'} \textit{ is an assignment to } R.fst\} \\
kill_{RD}([read\ R.snd]^l) &= \{(R.snd, ?)\} \cup \{(R.snd, l') | B^{l'} \textit{ is an assignment to } R.snd\} \\
kill_{RD}([write\ a]^l) &= \emptyset \\
kill_{RD}([b]^l) &= \emptyset
\end{aligned}
$$

$$
\begin{aligned}
gen_{RD}([int\ x]^l) &= \{x, l\} \\
gen_{RD}([int[n]\ A]^l) &= \{A, l\} \\
gen_{RD}([\{int\ fst;\ int\ snd\}\ R]^l) &= \{R.fst, l\} \cup \{R.snd, l\} \\
gen_{RD}([x := a]^l) &= \{x, l\} \\
gen_{RD}([A[a] := a']^l) &= \{A, l\} \\
gen_{RD}([R.fst := a]^l) &= \{R.fst, l\} \\
gen_{RD}([R.snd := a]^l) &= \{R.snd, l\} \\
gen_{RD}([R := (a_1, a_2)]^l) &= \{R.fst, l\} \cup \{R.snd, l\} \\
gen_{RD}([read\ x]^l) &= \{x, l\} \\
gen_{RD}([read\ A[a]]^l) &= \{A, l\} \\
gen_{RD}([read\ R.fst]^l) &= \{R.fst, l\} \\
gen_{RD}([read\ R.snd]^l) &= \{R.snd, l\} \\
gen_{RD}([write\ a]^l) &= \emptyset \\
gen_{RD}([b]^l) &= \emptyset
\end{aligned}
$$

2.1.3 Example Program

This section considers following example MICRO-C program, generating its flow graph and applying Reaching Definitions Analysis.

```
1 int x;                          <l1>
2 int[10] a;                      <l2>
3 {int fst; int snd} r;           <l3>
4 x := 10;                        <l4>
```

```
5  while x>0 {                          <l5>
6      if x==2 <l6> then x := x-1; <l7>
7      else x := x-2;               <l8>
8  }
9  x := 1;                          <l9>
10 a[x] := 3;                       <l10>
11 r.fst := a[x];                   <l11>
```

Flow graph of above program is shown in Figure 5.

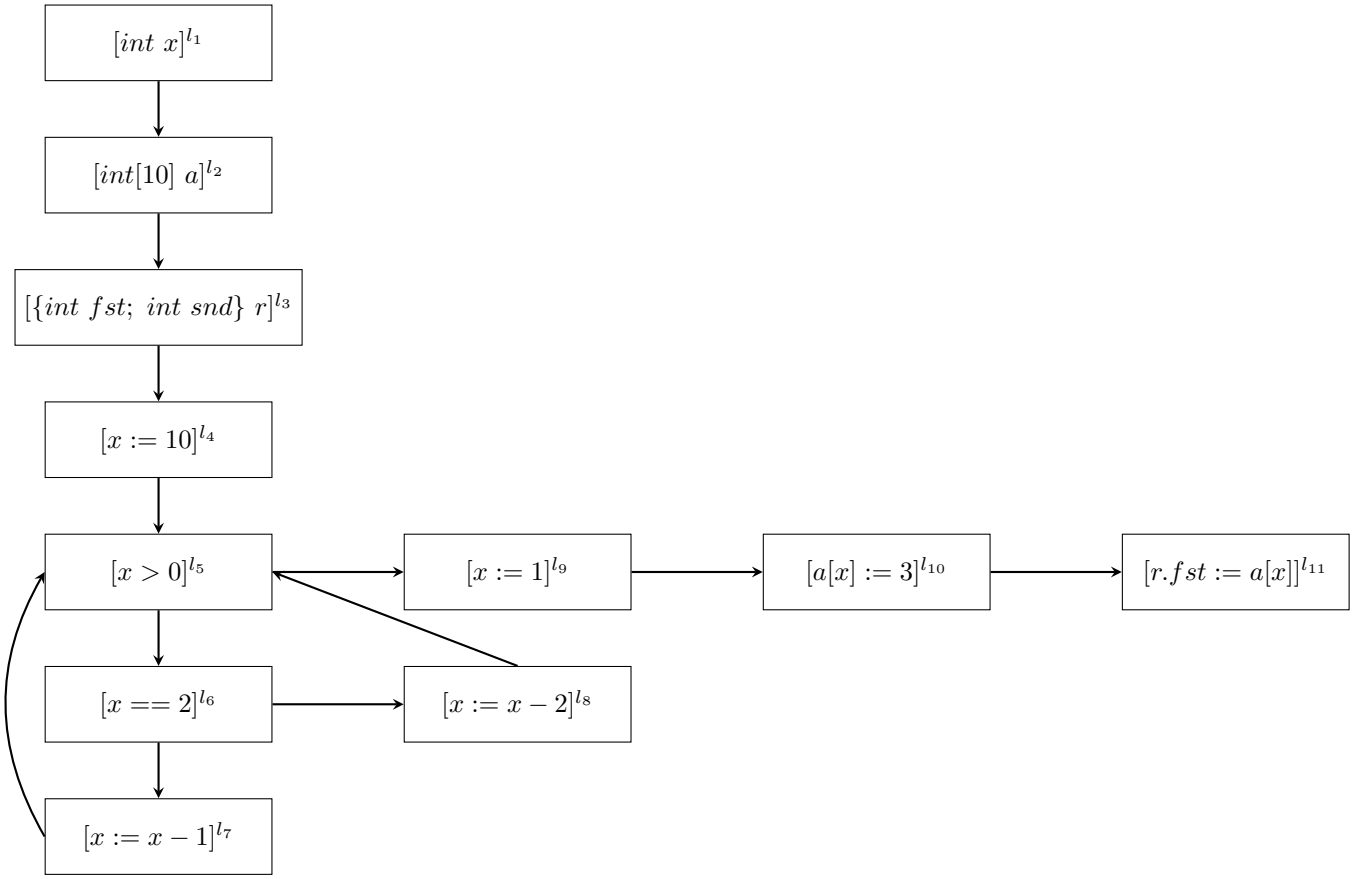


Figure 5: Flow graph for example MICRO-C program

Calculated $kill_{RD}$ and gen_{RD} sets and RD_0 and $RD_\bullet$ equations of above flow graph is shown in Table 2 and below.

l	$kill_{RD}(l)$	$gen_{RD}(l)$
1	$\{(x,?)\}$	$\{(x,1)\}$
2	$\{(a,?)\}$	$\{(a,2)\}$
3	$\{(r.fst,?),(r.snd,?)\}$	$\{(r.fst,3),(r.snd,3)\}$
4	$\{(x,?),(x,1),(x,4),(x,7),(x,8),(x,9)\}$	$\{(x,4)\}$
5	$\emptyset$	$\emptyset$
6	$\emptyset$	$\emptyset$
7	$\{(x,?),(x,1),(x,4),(x,7),(x,8),(x,9)\}$	$\{(x,7)\}$
8	$\{(x,?),(x,1),(x,4),(x,7),(x,8),(x,9)\}$	$\{(x,8)\}$
9	$\{(x,?),(x,1),(x,4),(x,7),(x,8),(x,9)\}$	$\{(x,9)\}$
10	$\{(a,?),(a,2),(a,10)\}$	$\{(a,10)\}$
11	$\{(r.fst,?),(r.fst,3),(r.fst,11)\}$	$\{(r.fst,11)\}$

Table 2: $kill_{RD}$ and gen_{RD} sets of example MICRO-C program

$$
\begin{aligned}
RD_0(1) &= \{(x,?),(a,?),(r.fst,?),(r.snd,?)\}\\
RD_0(2) &= RD_\bullet(1)\\
RD_0(3) &= RD_\bullet(2)\\
RD_0(4) &= RD_\bullet(3)\\
RD_0(5) &= RD_\bullet(4) \cup RD_\bullet(7) \cup RD_\bullet(8)\\
RD_0(6) &= RD_\bullet(5)\\
RD_0(7) &= RD_\bullet(6)\\
RD_0(8) &= RD_\bullet(6)\\
RD_0(9) &= RD_\bullet(5)\\
RD_0(10) &= RD_\bullet(9)\\
RD_0(11) &= RD_\bullet(10)
\end{aligned}
$$

$$
\begin{aligned}
RD_\bullet(1) &= (RD_0(1)\backslash\{(x,?)\}) \cup \{(x,1)\}\\
RD_\bullet(2) &= (RD_0(2)\backslash\{(a,?)\}) \cup \{(a,2)\}\\
RD_\bullet(3) &= (RD_0(3)\backslash\{(r.fst,?),(r.snd,?)\}) \cup \{(r.fst,3),(r.snd,3))\}\\
RD_\bullet(4) &= (RD_0(4)\backslash\{(x,?),(x,1),(x,4),(x,7),(x,8),(x,9)\}) \cup \{(x,4)\}\\
RD_\bullet(5) &= RD_0(5)\\
RD_\bullet(6) &= RD_0(6)\\
RD_\bullet(7) &= (RD_0(7)\backslash\{(x,?),(x,1),(x,4),(x,7),(x,8),(x,9)\}) \cup \{(x,7)\}\\
RD_\bullet(8) &= (RD_0(8)\backslash\{(x,?),(x,1),(x,4),(x,7),(x,8),(x,9)\}) \cup \{(x,8)\}\\
RD_\bullet(9) &= (RD_0(9)\backslash\{(x,?),(x,1),(x,4),(x,7),(x,8),(x,9)\}) \cup \{(x,9)\}\\
RD_\bullet(10) &= (RD_0(10)\backslash\{(a,?),(a,2),(a,10)\}) \cup \{(a,10)\}\\
RD_\bullet(11) &= (RD_0(11)\backslash\{(r.fst,?),(r.fst,3),(r.fst,11)\}) \cup \{(r.fst,11)\}
\end{aligned}
$$

After solving equations, we get following:

$$RD_0(1) = (x,?),(a,?),(r.fst,?),(r.snd,?)$$

$$
\begin{aligned}
RD_0(2) &= (a,?),(r.fst,?),(r.snd,?),(x,1)\\
RD_0(3) &= (r.fst,?),(r.snd,?),(x,1),(a,2)\\
RD_0(4) &= (x,1),(a,2),(r.fst,3),(r.snd,3)\\
RD_0(5) &= (a,2),(r.fst,3),(r.snd,3),(x,4),(x,7),(x,8)\\
RD_0(6) &= (a,2),(r.fst,3),(r.snd,3),(x,4),(x,7),(x,8)\\
RD_0(7) &= (a,2),(r.fst,3),(r.snd,3),(x,4),(x,7),(x,8)\\
RD_0(8) &= (a,2),(r.fst,3),(r.snd,3),(x,4),(x,7),(x,8)\\
RD_0(9) &= (a,2),(r.fst,3),(r.snd,3),(x,4),(x,7),(x,8)\\
RD_0(10) &= (a,2),(r.fst,3),(r.snd,3),(x,9)\\
RD_0(11) &= (r.fst,3),(r.snd,3),(x,9),(a,10)
\end{aligned}
$$

$$
\begin{aligned}
RD_\bullet(1) &= (a,?),(r.fst,?),(r.snd,?),(x,1)\\
RD_\bullet(2) &= (r.fst,?),(r.snd,?),(x,1),(a,2)\\
RD_\bullet(3) &= (x,1),(a,2),(r.fst,3),(r.snd,3)\\
RD_\bullet(4) &= (a,2),(r.fst,3),(r.snd,3),(x,4)\\
RD_\bullet(5) &= (a,2),(r.fst,3),(r.snd,3),(x,4),(x,7),(x,8)\\
RD_\bullet(6) &= (a,2),(r.fst,3),(r.snd,3),(x,4),(x,7),(x,8)\\
RD_\bullet(7) &= (a,2),(r.fst,3),(r.snd,3),(x,7)\\
RD_\bullet(8) &= (a,2),(r.fst,3),(r.snd,3),(x,8)\\
RD_\bullet(9) &= (a,2),(r.fst,3),(r.snd,3),(x,9)\\
RD_\bullet(10) &= (r.fst,3),(r.snd,3),(x,9),(a,10)\\
RD_\bullet(11) &= (r.snd,3),(x,9),(a,10),(r.fst,11)
\end{aligned}
$$

2.2 Live Variables Analysis

Live Variables Analysis determines for each program point, which variables may be live at the exit from the point.

Approach is first constructing the $kill_{LV}$ and gen_{LV} sets of reaching definitions for all the actions of the program. Next, constructing a set of equations defining the live variables at the various nodes of the flow graph and finally using an algorithm to solve the equations.

LV_0 and $LV_\bullet$ equations used in this analysis are below.

$$LV_\bullet = (LV_0 \backslash kill_{LV}(B^l)) \cup gen_{LV}(B^l)$$

$$LV_0 = \begin{cases} X_* & if\ l \in final(S_*) \\ \cup\{LV_\bullet(l') | (l',l) \in flow^R(S_*)\} & otherwise \end{cases}$$

where $flow^R(S_*) = \{(l',l) | (l,l') \in flow(S_*)\}$ and
X_* are live variables at the end of the program.

2.2.1 Handling Arrays and Records

As discussed in Section 2.1.1, array elements are reached with variables like $A[x] := 4$ or $y := A[x]$. This means that treating each element of array individually is not feasible since we cannot know which element of array A will be needed without execution. That is why we will treat arrays as live variables at every point in the program after they have been generated until they are declared.

Same discussion in Section 2.1.1 is also applicable for records. Handling $R.fst$ and $R.snd$ individually will not be an issue since we always know which element is being reached. That way we do not have to assume all elements of the record has changed which is a memory waste and pointless.

2.2.2 $kill_{LV}$ and gen_{LV} Sets For MICRO-C

$kill_{LV}$ and gen_{LV} sets for MICRO-C are as follows.

$$
\begin{aligned}
kill_{LV}([int\ x]^l) &= \{x\} \\
kill_{LV}([int[n]\ A]^l) &= \{A\} \\
kill_{LV}([\{int\ fst;\ int\ snd\}\ R]^l) &= \{R.fst,\ R.snd\} \\
kill_{LV}([x := a]^l) &= \{x\} \\
kill_{LV}([A[a] := a']^l) &= \emptyset \\
kill_{LV}([R.fst := a]^l) &= \{R.fst\} \\
kill_{LV}([R.snd := a]^l) &= \{R.snd\} \\
kill_{LV}([R := (a_1, a_2)]^l) &= \{R.fst,\ R.snd\} \\
kill_{LV}([read\ x]^l) &= \{x\} \\
kill_{LV}([read\ A[a]]^l) &= \emptyset \\
kill_{LV}([read\ R.fst]^l) &= \{R.fst\} \\
kill_{LV}([read\ R.snd]^l) &= \{R.snd\} \\
kill_{LV}([write\ a]^l) &= \emptyset \\
kill_{LV}([b]^l) &= \emptyset
\end{aligned}
$$

$$
\begin{aligned}
gen_{LV}([int\ x]^l) &= \emptyset \\
gen_{LV}([int[n]\ A]^l) &= \emptyset \\
gen_{LV}([\{int\ fst;\ int\ snd\}\ R]^l) &= \emptyset \\
gen_{LV}([x := a]^l) &= FV(a) \\
gen_{LV}([A[a] := a']^l) &= FV(a) \cup FV(a') \\
gen_{LV}([R.fst := a]^l) &= FV(a) \\
gen_{LV}([R.snd := a]^l) &= FV(a)
\end{aligned}
$$

$$
\begin{aligned}
gen_{LV}([R := (a_1, a_2)]^l) &= FV(a_1) \cup FV(a_2) \\
gen_{LV}([read\ x]^l) &= \emptyset \\
gen_{LV}([read\ A[a]]^l) &= FV(a) \\
gen_{LV}([read\ R.fst]^l) &= \emptyset \\
gen_{LV}([read\ R.snd]^l) &= \emptyset \\
gen_{LV}([write\ a]^l) &= FV(a) \\
gen_{LV}([b]^l) &= FV(b)
\end{aligned}
$$

The reason why declarations are killing elements that being declared is because we assume that when a variable is declared, it is value is automatically set to zero.

2.2.3 Example Program

This section considers the example MICRO-C program in Section 2.1.3, generating its flow graph and applying Live Variables Analysis.

Live Variables Analysis is a backwards analysis so reverse flow graph $flow^R(S_*)$ is used and it is shown in Figure 6.

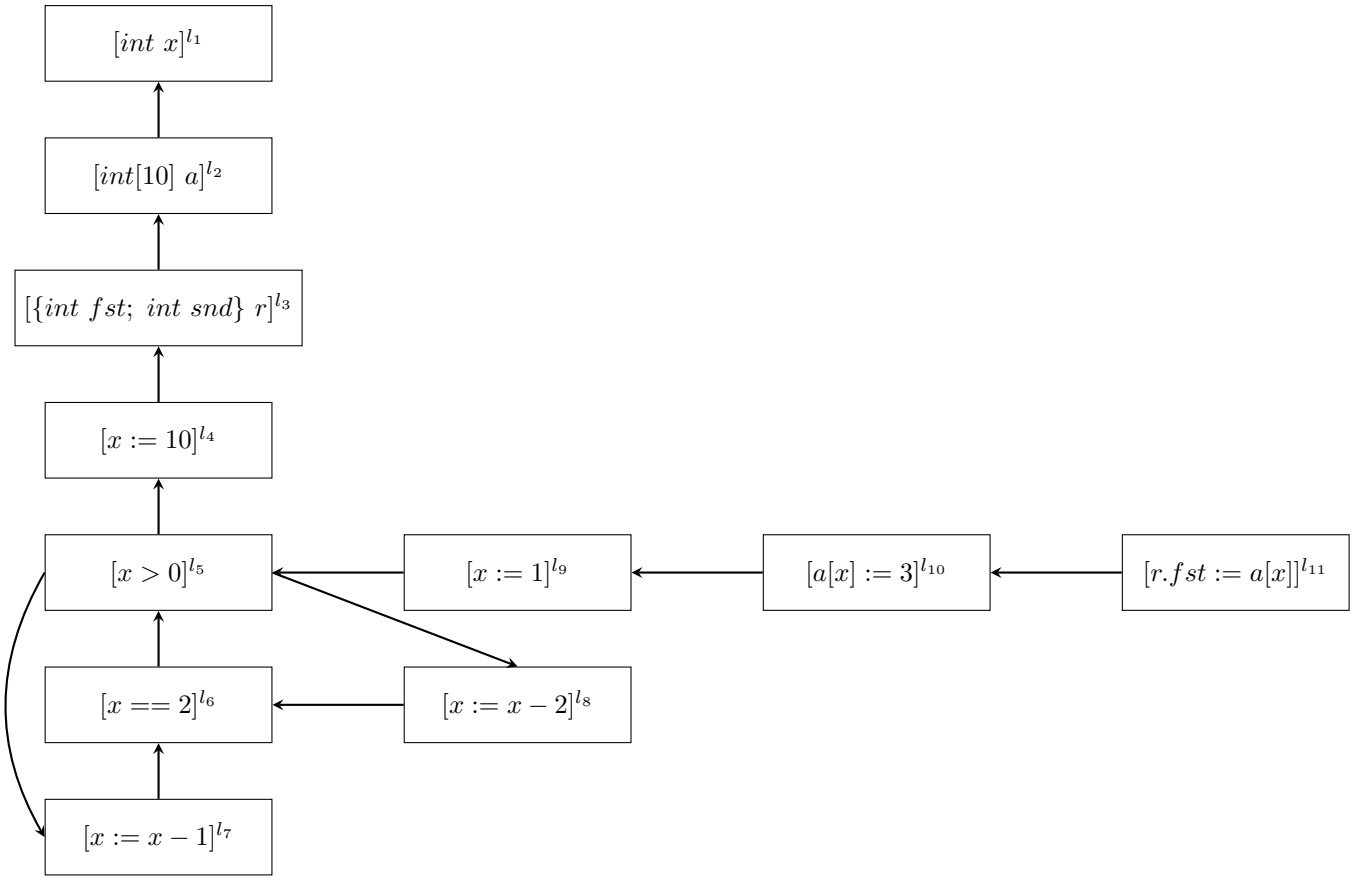


Figure 6: $Flow^R$ graph for example MICRO-C program

Calculated $kill_{LV}$ and gen_{LV} sets and LV_0 and $LV_\bullet$ equations of above flow graph is shown

in Table 3 and below.

l	$kill_{LV}(l)$	$gen_{LV}(l)$
1	$\{x\}$	$\emptyset$
2	$\{a\}$	$\emptyset$
3	$\{r.fst, r.snd\}$	$\emptyset$
4	$\{x\}$	$\emptyset$
5	$\emptyset$	$\{x\}$
6	$\emptyset$	$\{x\}$
7	$\{x\}$	$\{x\}$
8	$\{x\}$	$\{x\}$
9	$\{x\}$	$\emptyset$
10	$\emptyset$	$\emptyset$
11	$\{r.fst\}$	$\{x, a\}$

Table 3: $kill_{LV}$ and gen_{LV} sets of example MICRO-C program

$$
\begin{aligned}
LV_0(1) &= LV_\bullet(2)\\
LV_0(2) &= LV_\bullet(3)\\
LV_0(3) &= LV_\bullet(4)\\
LV_0(4) &= LV_\bullet(5)\\
LV_0(5) &= LV_\bullet(6) \cup LV_\bullet(9)\\
LV_0(6) &= LV_\bullet(7) \cup LV_\bullet(8)\\
LV_0(7) &= LV_\bullet(5)\\
LV_0(8) &= LV_\bullet(5)\\
LV_0(9) &= LV_\bullet(10)\\
LV_0(10) &= LV_\bullet(11)\\
LV_0(11) &= \{r.fst\}
\end{aligned}
$$

$$
\begin{aligned}
LV_\bullet(1) &= (LV_0(1)\backslash\{x\})\\
LV_\bullet(2) &= (LV_0(2)\backslash\{a\})\\
LV_\bullet(3) &= (LV_0(3)\backslash\{r.fst, r.snd\})\\
LV_\bullet(4) &= (LV_0(4)\backslash\{x\})\\
LV_\bullet(5) &= LV_0(5) \cup \{x\}\\
LV_\bullet(6) &= LV_0(6) \cup \{x\}\\
LV_\bullet(7) &= (LV_0(7)\backslash\{x\}) \cup \{x\}\\
LV_\bullet(8) &= (LV_0(8)\backslash\{x\}) \cup \{x\}\\
LV_\bullet(9) &= (LV_0(9)\backslash\{x\})\\
LV_\bullet(10) &= LV_0(10)\\
LV_\bullet(11) &= (LV_0(11)\backslash\{r.fst\}) \cup \{x, a\}
\end{aligned}
$$

After solving equations, we get following:

$$
\begin{aligned}
LV_0(11) &= \{r.fst\} \\
LV_0(10) &= \{x, a\} \\
LV_0(9) &= \{x, a\} \\
LV_0(8) &= \{x, a\} \\
LV_0(7) &= \{x, a\} \\
LV_0(6) &= \{x, a\} \\
LV_0(5) &= \{x, a\} \\
LV_0(4) &= \{x, a\} \\
LV_0(3) &= \{a\} \\
LV_0(2) &= \{a\} \\
LV_0(1) &= \emptyset
\end{aligned}
$$

$$
\begin{aligned}
LV_\bullet(11) &= \{x, a\} \\
LV_\bullet(10) &= \{x, a\} \\
LV_\bullet(9) &= \{a\} \\
LV_\bullet(8) &= \{x, a\} \\
LV_\bullet(7) &= \{x, a\} \\
LV_\bullet(6) &= \{x, a\} \\
LV_\bullet(5) &= \{x, a\} \\
LV_\bullet(4) &= \{a\} \\
LV_\bullet(3) &= \{a\} \\
LV_\bullet(2) &= \emptyset \\
LV_\bullet(1) &= \emptyset
\end{aligned}
$$

2.3 Dangerous Variables Analysis

The Reaching Definitions Analysis can be used to determine program points where a variable might be dangerous meaning that its value might depend on the initial memory directly or indirectly. A modification of the analysis called Dangerous Variables Analysis determines this information directly.

To modify RDA, we need to remove labels from *kill* and *gen* sets. The major modification to RDA is based on the fact that in order to generate dangerous variables, for assignments we need to be able check previous steps. As an example, for program piece $x := y$, if y is a dangerous variable, then x will be generated. But if y is not a dangerous variable, then x will be killed. To be able to solve this issue, we are going to give DV_0 sets of the labels to *kill* and *gen* functions as a parameter.

DV_0 and $DV_\bullet$ equations used in this analysis are below.

$$DV_\bullet = (DV_0 \backslash kill_{DV}(B^l, DV_0)) \cup gen_{DV}(B^l, DV_0)$$

$$DV_0 = \begin{cases} \{x | x \in FV(S_*)\} & if\ l = init(S_*) \\ \cup \{DV_\bullet(l') | (l', l) \in flow(S_*)\} & otherwise \end{cases}$$

2.3.1 $kill_{DV}$ and gen_{DV} Sets For MICRO-C

$kill_DV$ and gen_DV sets for MICRO-C are as follows.

$$\begin{aligned}
kill_{DV}([int\ x]^l, DV_0) &= \{x\} \\
kill_{DV}([int[n]\ A]^l, DV_0) &= \{A\} \\
kill_{DV}([\{int\ fst;\ int\ snd\}\ R]^l, DV_0) &= \{R.fst, R.snd\} \\
kill_{DV}([x := a]^l, DV_0) &= \begin{cases} \{x\} & if\ a \notin DV_0 \\ \emptyset & otherwise \end{cases} \\
kill_{DV}([A[a] := a']^l, DV_0) &= \begin{cases} \{A\} & if\ a' \notin DV_0 \\ \emptyset & otherwise \end{cases} \\
kill_{DV}([R.fst := a]^l, DV_0) &= \begin{cases} \{R.fst\} & if\ a \notin DV_0 \\ \emptyset & otherwise \end{cases} \\
kill_{DV}([R.snd := a]^l, DV_0) &= \begin{cases} \{R.snd\} & if\ a \notin DV_0 \\ \emptyset & otherwise \end{cases} \\
kill_{DV}([R := (a_1, a_2)]^l, DV_0) &= \begin{cases} \{R.fst\} & if\ a_1 \notin DV_0 \\ \{R.snd\} & if\ a_2 \notin DV_0 \\ \{R.fst, R.snd\} & if\ a_1 \notin DV_0\ \wedge\ a_2 \notin DV_0 \\ \emptyset & otherwise \end{cases} \\
kill_{DV}([read\ x]^l, DV_0) &= \{x\} \\
kill_{DV}([read\ A[a]]^l, DV_0) &= \{A\} \\
kill_{DV}([read\ R.fst]^l, DV_0) &= \{R.fst\} \\
kill_{DV}([read\ R.snd]^l, DV_0) &= \{R.snd)\} \\
kill_{DV}([write\ a]^l, DV_0) &= \emptyset \\
kill_{DV}([b]^l, DV_0) &= \emptyset
\end{aligned}$$

$$\begin{aligned}
gen_{DV}([int\ x]^l, DV_0) &= \emptyset \\
gen_{DV}([int[n]\ A]^l, DV_0) &= \emptyset \\
gen_{DV}([\{int\ fst;\ int\ snd\}\ R]^l, DV_0) &= \emptyset \\
gen_{DV}([x := a]^l, DV_0) &= \begin{cases} \{x\} & if\ a \in DV_0 \\ \emptyset & otherwise \end{cases} \\
gen_{DV}([A[a] := a']^l, DV_0) &= \{A\} \quad if\ a' \in DV_0
\end{aligned}$$

$$
\begin{aligned}
& & \emptyset & \quad otherwise \\
gen_{DV}([R.fst := a]^l, DV_0) = & & \{R.fst\} & \quad if\ a \in DV_0 \\
& & \emptyset & \quad otherwise \\
gen_{DV}([R.snd := a]^l, DV_0) = & & \{R.snd\} & \quad if\ a \in DV_0 \\
& & \emptyset & \quad otherwise \\
gen_{DV}([R := (a_1, a_2)]^l, DV_0) = & & \{R.fst\} & \quad if\ a_1 \in DV_0 \\
& & \{R.snd\} & \quad if\ a_2 \in DV_0 \\
& & \{R.fst, R.snd\} & \quad if\ a_1 \in DV_0\ \wedge\ a_2 \in DV_0 \\
& & \emptyset & \quad otherwise \\
gen_{DV}([read\ x]^l, DV_0) = & & \emptyset & \\
gen_{DV}([read\ A[a]]^l, DV_0) = & & \emptyset & \\
gen_{DV}([read\ R.fst]^l, DV_0) = & & \emptyset & \\
gen_{DV}([read\ R.snd]^l, DV_0) = & & \emptyset & \\
gen_{DV}([write\ a]^l, DV_0) = & & \emptyset & \\
gen_{DV}([b]^l, DV_0) = & & \emptyset &
\end{aligned}
$$

2.4 Faint Variables Analysis

The Live Variables Analysis can be used to determine program points where a variables might be faint meaning that its value is not needed directly or indirectly before the program terminates. A modification of the analysis called Faint Variables Analysis determines this information directly.

To modify LVA, X_* will be the faint variables at the end of the program. The major modification to LVA is based on the fact that in order to generate faint variables, for assignments we need to be able check previous steps. As an example, for program piece $x := y$, if x is a faint variable, then nothing will be generated or killed. But if x is not a faint variable, then x will be generated and y will be killed. To be able to solve this issue, we are going to give FAV_0 sets of the labels to *kill* and *gen* functions as a parameter.

FAV_0 and $FAV_\bullet$ equations used in this analysis are below.

$$FAV_\bullet = (FAV_0 \backslash kill_{FAV}(B^l, FAV_0)) \cup gen_{FAV}(B^l, FAV_0)$$

$$FAV_0 = \begin{cases} X_* & if\ l \in final(S_*) \\ \cup\{FAV_\bullet(l') | (l', l) \in flow^R(S_*)\} & otherwise \end{cases}$$

where $flow^R(S_*) = \{(l', l) | (l, l') \in flow(S_*)\}$ and
X_* are the faint variables at the end of the program.

2.4.1 $kill_{FAV}$ and gen_{FAV} Sets For MICRO-C

$kill_{FAV}$ and gen_{FAV} sets for MICRO-C are as follows.

$$kill_{FAV}([int\ x]^l, FAV_0) = \emptyset$$

$$
\begin{aligned}
kill_{FAV}([int[n]\ A]^l, FAV_0) &= \emptyset \\
kill_{FAV}([\{int\ fst;\ int\ snd\}\ R]^l, FAV_0) &= \emptyset \\
kill_{FAV}([x := a]^l, FAV_0) &= FV(a) \quad if\ x\ \notin FAV_0 \\
&\quad\ \emptyset \quad otherwise \\
kill_{FAV}([A[a] := a']^l, FAV_0) &= FV(a) \quad if\ A\ \notin FAV_0 \\
&\quad\ \emptyset \quad otherwise \\
kill_{FAV}([R.fst := a]^l, FAV_0) &= FV(a) \quad if\ R.fst\ \notin FAV_0 \\
&\quad\ \emptyset \quad otherwise \\
kill_{FAV}([R.snd := a]^l, FAV_0) &= FV(a) \quad if\ R.snd\ \notin FAV_0 \\
&\quad\ \emptyset \quad otherwise \\
kill_{FAV}([R := (a_1, a_2)]^l, FAV_0) &= FV(a_1) \quad if\ R.fst \notin FAV_0 \\
&\quad\ FV(a_2) \quad if\ R.snd \notin FV_0 \\
&\quad\ FV(a_1) \cup FV(a_2) \quad if\ R.fst \notin FAV_0\ \wedge\ R.snd \notin FAV_0 \\
&\quad\ \emptyset \quad otherwise
\end{aligned}
$$

$$
\begin{aligned}
kill_{FAV}([read\ x]^l, FAV_0) &= \emptyset \\
kill_{FAV}([read\ A[a]]^l, FAV_0) &= \emptyset \\
kill_{FAV}([read\ R.fst]^l, FAV_0) &= \emptyset \\
kill_{FAV}([read\ R.snd]^l, FAV_0) &= \emptyset \\
kill_{FAV}([write\ a]^l, FAV_0) &= FV(a) \\
kill_{FAV}([b]^l, FAV_0) &= FV(b)
\end{aligned}
$$

$$
\begin{aligned}
gen_{FAV}([int\ x]^l, FAV_0) &= \emptyset \\
gen_{FAV}([int[n]\ A]^l, FAV_0) &= \emptyset \\
gen_{FAV}([\{int\ fst;\ int\ snd\}\ R]^l, FAV_0) &= \emptyset \\
gen_{FAV}([x := a]^l, FAV_0) &= \{x\} \\
gen_{FAV}([A[a] := a']^l, FAV_0) &= \emptyset \\
gen_{FAV}([R.fst := a]^l, FAV_0) &= \{R.fst\} \\
gen_{FAV}([R.snd := a]^l, FAV_0) &= \{R.snd\} \\
gen_{FAV}([R := (a_1, a_2)]^l, FAV_0) &= \{R.fst, R.snd\} \\
gen_{FAV}([read\ x]^l, FAV_0) &= \{x\} \\
gen_{FAV}([read\ A[a]]^l, FAV_0) &= \emptyset \\
gen_{FAV}([read\ R.fst]^l, FAV_0) &= \{R.fst\} \\
gen_{FAV}([read\ R.snd]^l, FAV_0) &= \{R.snd\} \\
gen_{FAV}([write\ a]^l, FAV_0) &= \emptyset
\end{aligned}
$$

$$gen_{FAV}([b]^l, FAV_0) = \emptyset$$

2.5 Detection of Signs Analysis

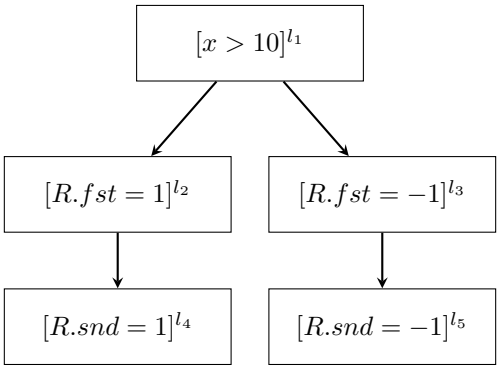
Detection of Signs Analysis determines for each program point and each variable, the possible signs (negative, zero or positive) that it may have whenever execution reaches that point.

2.5.1 Handling Arrays and Records

As discussed in Section 2.1.1, array elements are reached with variables like $A[x] := 4$ or $y := A[x]$. This means that treating each element of array individually is not feasible since we cannot know which element of array A will be needed without execution. We chose again to handle arrays as single variables. An interesting issue about the arrays with Detection of Signs Analysis is that we can detect negative indexing. If we encounter a negative index algorithm can detect it and then we can warn the programmer.

Same discussion in Section 2.1.1 is also applicable for records. Handling $R.fst$ and $R.snd$ individually will not be an issue since we always know which element is being reached. That way we do not have to assume all elements of the record has changed.

One can include relationship between signs of the $R.fst$ and $R.snd$. An example flow graph would be below:



After l_4 and l_5, one would get 4 possibilities if $R.fst$ and $R.snd$ are handled individually as two different variables ((+,+),(-,-),(+,-),(-,+)). But we know that there are only two possibilities (+,+),(-,-). In order to capture that, analysis can handle records as pair individuals like $R = (R.fst, R.snd)$. We will not go with that route to simplify our algorithm. So we will handle records as we did before, $R.fst$ and $R.snd$ will be individual two different variables.

2.5.2 Analysis Domain and Function

Detection of Signs Analysis can be specified as a monotone framework as follows.

Definition of a complete lattice:

- Analysis domain: $L = \widehat{State_S} = (Var_* \to \mathcal{P}(\{-, 0, +\}), \sqsubseteq)$
  - $\perp \in Var_* \to \mathcal{P}(\{-, 0, +\}), \perp(x) = \emptyset \quad \forall x \in Var_*$
  - $\top \in Var_* \to \mathcal{P}(\{-, 0, +\}), \top(x) = \{0, -, +\} \quad \forall x \in Var_*$

- $\mathcal{F} = \{f^s : L \to L \mid f^s \ is \ monotone\}$
- Analysis functions f^s are shown in Table 4.
- Flow graph: $F = flow(S_*)$
- Labels: $E = \{init(S_*)\}$
- Initial element: $\iota(x) = \{-, 0, +\} \ for \ all \ variables \ x$

$Block^l$	$f^s_l(\widehat{\sigma})$
$[int\ x]^l$	$\begin{cases} \widehat{\sigma}[x \mapsto \{0\}] & if\ \widehat{\sigma} \neq \bot \\ \bot & otherwise \end{cases}$
$[int[n]\ A]^l$	$\begin{cases} \widehat{\sigma}[A \mapsto \{0\}] & if\ n > 0\ and\ \widehat{\sigma} \neq \bot \\ \bot & otherwise \end{cases}$
$[\{int\ fst;\ int\ snd\}R]^l$	$\begin{cases} \widehat{\sigma}[R.fst \mapsto \{0\}, R.snd \mapsto \{0\}] & if\ \widehat{\sigma} \neq \bot \\ \bot & otherwise \end{cases}$
$[x := a]^l$	$\begin{cases} \widehat{\sigma}[x \mapsto A_S[\![a]\!]\widehat{\sigma}] & if\ \widehat{\sigma} \neq \bot \\ \bot & otherwise \end{cases}$
$[A[a] := a']^l$	$\begin{cases} \bot & if A_S[\![a]\!]\widehat{\sigma} \cap \{0, +\} = \{\}\ or\ \widehat{\sigma} = \bot \\ \widehat{\sigma}[A \mapsto A_S[\![a']\!]\widehat{\sigma} \cup \widehat{\sigma}(A)] & otherwise \end{cases}$
$[R.fst := a]^l$	$\begin{cases} \widehat{\sigma}[R.fst \mapsto A_S[\![a]\!]\widehat{\sigma}] & if\ \widehat{\sigma} \neq \bot \\ \bot & otherwise \end{cases}$
$[R.snd := a]^l$	$\begin{cases} \widehat{\sigma}[R.snd \mapsto A_S[\![a]\!]\widehat{\sigma}] & if\ \widehat{\sigma} \neq \bot \\ \bot & otherwise \end{cases}$
$[R := (a_1, a_2)]^l$	$\begin{cases} \widehat{\sigma}[R.fst \mapsto A_S[\![a_1]\!]\widehat{\sigma}, R.snd \mapsto A_S[\![a_2]\!]\widehat{\sigma}] & if\ \widehat{\sigma} \neq \bot \\ \bot & otherwise \end{cases}$
$[read\ x]^l$	$\begin{cases} \widehat{\sigma}[x \mapsto \{-, 0, +\}] & if\ \widehat{\sigma} \neq \bot \\ \bot & otherwise \end{cases}$
$[read\ A[a]]^l$	$\begin{cases} \bot & if A_S[\![a]\!]\widehat{\sigma} \cap \{0, +\} = \{\}\ or\ \widehat{\sigma} = \bot \\ \widehat{\sigma}[A \mapsto \{-, 0, +\}] & otherwise \end{cases}$
$[read\ R.fst]^l$	$\begin{cases} \widehat{\sigma}[R.fst \mapsto \{-, 0, +\}] & if\ \widehat{\sigma} \neq \bot \\ \bot & otherwise \end{cases}$
$[read\ R.snd]^l$	$\begin{cases} \widehat{\sigma}[R.snd \mapsto \{-, 0, +\}] & if\ \widehat{\sigma} \neq \bot \\ \bot & otherwise \end{cases}$
$[write\ a]^l$	$\widehat{\sigma}$
$[write\ A[a]]^l$	$\begin{cases} \bot & if A_S[\![a]\!]\widehat{\sigma} \subseteq \{-\}\ or\ \widehat{\sigma} = \bot \\ \widehat{\sigma} & otherwise \end{cases}$
$[b]^l$	$\widehat{\sigma}$

Table 4: Transfer functions for MICRO-C language

$A_S : AExp \to (\widehat{State_S} \to \mathcal{P}(\{-, 0, +\}))$ determines the signs of arithmetic expressions:

$$A_S[\![n]\!]\widehat{\sigma} = \begin{cases} \{+\} & if\ n > 0 \\ \{0\} & if\ n = 0 \\ \{-\} & if\ n < 0 \end{cases}$$

$$
\begin{aligned}
A_S[\![x]\!]\widehat{\sigma} &= \widehat{\sigma}(x) \\
A_S[\![A[a]]\!]\widehat{\sigma} &= \begin{cases} \bot & if\, A_S[\![a]\!]\widehat{\sigma} \cap \{0,+\} = \{\} \\ \widehat{\sigma}(A) & otherwise \end{cases} \\
A_S[\![R.fst]\!]\widehat{\sigma} &= \widehat{\sigma}(R.fst) \\
A_S[\![R.snd]\!]\widehat{\sigma} &= \widehat{\sigma}(R.snd) \\
A_S[\![a_1 \ \widehat{op_a}\ a_2]\!]\widehat{\sigma} &= A_S[\![a_1]\!]\widehat{\sigma}\ \widehat{op_a}\ A_S[\![a_2]\!]\widehat{\sigma}
\end{aligned}
$$

$\widehat{op_a} = +$	-	0	+
-	{-}	{-}	{-, 0, +}
0	{-}	{0}	{+}
+	{-, 0, +}	{+}	{+}

$\widehat{op_a} = *$	-	0	+
-	{+}	{0}	{-}
0	{0}	{0}	{0}
+	{-}	{0}	{+}

$\widehat{op_a} = -$	-	0	+
-	{-, 0, +}	{-}	{-}
0	{+}	{0}	{-}
+	{+}	{+}	{-, 0, +}

$\widehat{op_a} = /$	-	0	+
-	{0, +}	$\bot$	{-}
0	{0}	$\bot$	{0}
+	{-}	$\bot$	{0, +}

In order to show that analysis functions are monotone, we will prove it on assignment function $f_l^S(\widehat{\sigma}) = \widehat{\sigma}[x \mapsto A_S[\![a]\!]\widehat{\sigma}]$.

First we need to show that function $A_S[\![a]\!]\widehat{\sigma}$ is monotone. There are 5 possibilities for a $(n, x, A[a'], R.fst, R.snd$ and $a_1\ \widehat{op_a}\ a_2)$. We will only show n, x and $a_1\ \widehat{op_a}\ a_2$ since $A[a'], R.fst$ and $R.snd$ is the same as x.

- $a : n$
  $\widehat{\sigma_1} \sqsubseteq \widehat{\sigma_2} \Rightarrow A_S[\![n]\!]\widehat{\sigma_1} \sqsubseteq A_S[\![n]\!]\widehat{\sigma_2}$
  - $n > 0$
    $$A_S[\![n]\!]\widehat{\sigma_1} = A_S[\![n]\!]\widehat{\sigma_2} = \{+\}$$
    $$A_S[\![n]\!]\widehat{\sigma_1} \sqsubseteq A_S[\![n]\!]\widehat{\sigma_2}$$
  - $n = 0$
    $$A_S[\![n]\!]\widehat{\sigma_1} = A_S[\![n]\!]\widehat{\sigma_2} = \{0\}$$
    $$A_S[\![n]\!]\widehat{\sigma_1} \sqsubseteq A_S[\![n]\!]\widehat{\sigma_2}$$
  - $n < 0$
    $$A_S[\![n]\!]\widehat{\sigma_1} = A_S[\![n]\!]\widehat{\sigma_2} = \{-\}$$
    $$A_S[\![n]\!]\widehat{\sigma_1} \sqsubseteq A_S[\![n]\!]\widehat{\sigma_2}$$
- $a : x$
  $\widehat{\sigma_1} \sqsubseteq \widehat{\sigma_2} \Rightarrow A_S[\![x]\!]\widehat{\sigma_1} \sqsubseteq A_S[\![x]\!]\widehat{\sigma_2}$
  $$A_S[\![x]\!]\widehat{\sigma_1} = A_S[\![x]\!]\widehat{\sigma_2} = \widehat{\sigma}(x)$$
  $$A_S[\![x]\!]\widehat{\sigma_1} \sqsubseteq A_S[\![x]\!]\widehat{\sigma_2}$$

- $a : a_1 \ \widehat{op_a} \ a_2$
  $\widehat{\sigma_1} \sqsubseteq \widehat{\sigma_2} \Rightarrow A_S[\![a_1 \ \widehat{op_a} \ a_2]\!]\widehat{\sigma_1} \sqsubseteq A_S[\![a_1 \ \widehat{op_a} \ a_2]\!]\widehat{\sigma_2}$

$$A_S[\![a_1]\!]\widehat{\sigma_1} \sqsubseteq A_S[\![a_1]\!]\widehat{\sigma_2}$$
$$A_S[\![a_2]\!]\widehat{\sigma_1} \sqsubseteq A_S[\![a_2]\!]\widehat{\sigma_2}$$
$$A_S[\![a_1]\!]\widehat{\sigma_1} \ \widehat{op_a} \ A_S[\![a_2]\!]\widehat{\sigma_1} \sqsubseteq A_S[\![a_1]\!]\widehat{\sigma_2} \ \widehat{op_a} \ A_S[\![a_2]\!]\widehat{\sigma_2}$$
$$A_S[\![a_1 \ \widehat{op_a} \ a_2]\!]\widehat{\sigma_1} \sqsubseteq A_S[\![a_1 \ \widehat{op_a} \ a_2]\!]\widehat{\sigma_2}$$

Now having proven that $A_S[\![a]\!]\widehat{\sigma}$ is indeed monotone, we can also prove that $f_l^S(\widehat{\sigma}) = \widehat{\sigma}[x \mapsto A_S[\![a]\!]\widehat{\sigma}]$ is also monotone for assignments.

$$\widehat{\sigma_1} \sqsubseteq \widehat{\sigma_2} \Rightarrow \widehat{\sigma}[x \mapsto A_S[\![a]\!]\widehat{\sigma_1}] \sqsubseteq \widehat{\sigma}[x \mapsto A_S[\![a]\!]\widehat{\sigma_2}]$$
$$A_S[\![a]\!]\widehat{\sigma_1} \sqsubseteq A_S[\![a]\!]\widehat{\sigma_2}$$
$$x \mapsto A_S[\![a]\!]\widehat{\sigma_1} \sqsubseteq x \mapsto A_S[\![a]\!]\widehat{\sigma_2}$$
$$\widehat{\sigma}[x \mapsto A_S[\![a]\!]\widehat{\sigma_1}] \sqsubseteq \widehat{\sigma}[x \mapsto A_S[\![a]\!]\widehat{\sigma_2}]$$

2.5.3 Example Program

This section considers following MICRO-C program for Detection of Signs Analysis.

```
1 int x;                              <l1>
2 int y;                              <l2>
3 int[10] a;                          <l3>
4 x := 3;                             <l4>
5 y := -2;                            <l5>
6 if x>2 <l6> then x := x-1;          <l7>
7 else x := x+1;                      <l8>
8 a[x] := y+y;                        <l9>
```

Flow graph of above program is shown in Figure 7.

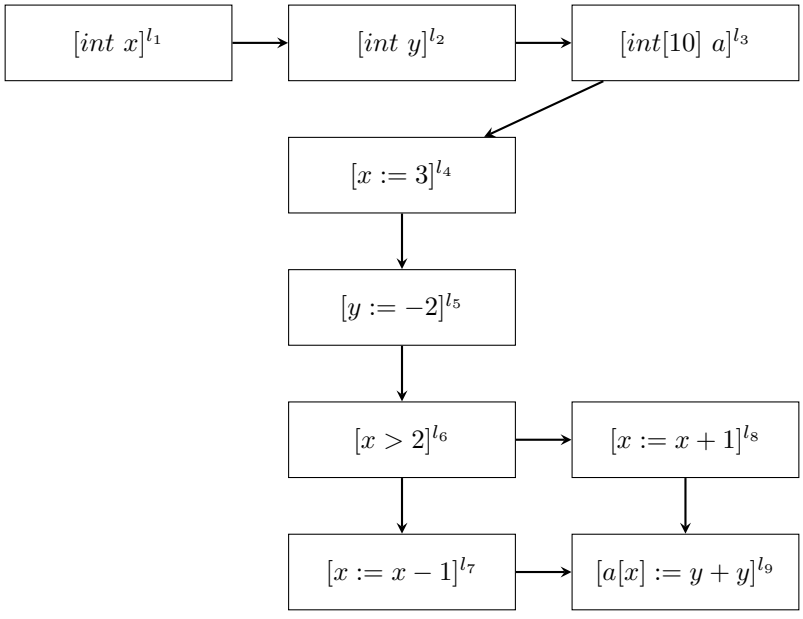


Figure 7: Flow graph for example MICRO-C program

Initial element: $\iota = [x \mapsto \{-, 0, +\}, y \mapsto \{-, 0, +\}, a \mapsto \{-, 0, +\}]$
Equations:

$$
\begin{aligned}
A_\bullet(1) &= f^s_{int\ x}(\iota) \\
A_\bullet(2) &= f^s_{int\ y}(A_\bullet(1)) \\
A_\bullet(3) &= f^s_{int[10]\ a}(A_\bullet(2)) \\
A_\bullet(4) &= f^s_{x:=3}(A_\bullet(3)) \\
A_\bullet(5) &= f^s_{y:=-2}(A_\bullet(4)) \\
A_\bullet(6) &= f^s_{x>2}(A_\bullet(5)) \\
A_\bullet(7) &= f^s_{x:=x-1}(A_\bullet(6)) \\
A_\bullet(8) &= f^s_{x:=x+1}(A_\bullet(6)) \\
A_\bullet(9) &= f^s_{a[x]:=y*y}(A_\bullet(7) \cup A_\bullet(8))
\end{aligned}
$$

Solution:

$A_\bullet(.)$	x	y	a
1	{0}	{-,0,+}	{-,0,+}
2	{0}	{0}	{-,0,+}
3	{0}	{0}	{0}
4	{+}	{0}	{0}
5	{+}	{-}	{0}
6	{+}	{-}	{0}
7	{-, 0, +}	{-}	{0}
8	{+}	{-}	{0}
9	{-, 0, +}	{-}	{-, 0}

2.5.4 Improvements

As it can be seen from the transfer functions on Table 4, we haven't dealt with boolean expressions. One can use the result of the tests to pass different information of the branches of the boolean expressions.

To improve the analysis we can change analysis function for boolean expressions: changing labeling $[b]^l$ to $[b]^{l_t,l_f}$, letting f_{l_t} be the analysis function for the true branch and f_{l_f} be the analysis function for the false branch.

Two analysis functions for $[b]^{l_t,l_f}$ then would be:

$$
\begin{aligned}
f_{l_t}(\widehat{\sigma}) &= \sqcup\{\widehat{\sigma'} \in Basic(\widehat{\sigma}) \mid tt \in \mathcal{B}_S[\![b]\!]\widehat{\sigma'}\} \\
f_{l_f}(\widehat{\sigma}) &= \sqcup\{\widehat{\sigma'} \in Basic(\widehat{\sigma}) \mid ff \in \mathcal{B}_S[\![b]\!]\widehat{\sigma'}\}
\end{aligned}
$$

where $Basic(\widehat{\sigma}) = \{\widehat{\sigma'} \sqsubseteq \widehat{\sigma} \mid \forall x :\ |\widehat{\sigma'}(x)| = 1\}$.

$\mathcal{B}_S : BExp \to (\widehat{State_S} \to \mathcal{P}(\{tt, ff\}))$ determine the signs of boolean expressions:

$$
\begin{aligned}
\mathcal{B}_S[\![a_1\ op_r\ a_2]\!]\widehat{\sigma'} &= A_S[\![a_1]\!]\widehat{\sigma'}\ \widehat{op_r}\ A_S[\![a_2]\!]\widehat{\sigma'} \\
\mathcal{B}_S[\![b_1\ op_b\ b_2]\!]\widehat{\sigma'} &= \mathcal{B}_S[\![b_1]\!]\widehat{\sigma'}\ \widehat{op_b}\ \mathcal{B}_S[\![b_2]\!]\widehat{\sigma'} \\
\mathcal{B}_S[\![\neg b]\!]\widehat{\sigma'} &= \{\neg t \mid t \in \mathcal{B}_s[\![b]\!]\widehat{\sigma'}\}
\end{aligned}
$$

Two examples ($\widehat{op_r} = \geq$ and $\widehat{op_r} = \leq$):

$\widehat{op_r} = \geq$	-	0	+
-	{tt,ff}	{ff}	{ff}
0	{tt}	{tt}	{ff}
+	{tt}	{tt}	{tt,ff}

$\widehat{op_r} = \leq$	-	0	+
-	{tt,ff}	{tt}	{tt}
0	{ff}	{tt}	{tt}
+	{ff}	{ff}	{tt,ff}

2.6 Interval Analysis

Interval Analysis determines for each program point and each variable the analysis determines a lower bound and an upper bound of the values that the variable will have whenever the execution reaches that point.

2.6.1 Handling Arrays and Records

As discussed in Section 2.1.1, array elements are reached with variables like $A[x] := 4$ or $y := A[x]$. This means that treating each element of array individually is not feasible since we cannot know which element of array A will be needed without execution. We chose again to handle arrays as single variables. An interesting issue about the arrays with Interval Analysis is that we can detect negative indexing. We can check if index is in interval $[-\infty, -1]$ and warn the programmer.

Same discussion in Section 2.1.1 is also applicable for records. Handling $R.fst$ and $R.snd$ individually will not be an issue since we always know which element is being reached. That way we do not have to assume all elements of the record has changed.

2.6.2 Analysis Domain and Function

Interval Analysis can be specified as a monotone framework as follows.

Definition of a complete lattice:

- Analysis domain: $L = \widehat{State_I} = (Var_* \to Interval, \sqsubseteq)$
  - $Interval = \{\bot\} \cup \{[z_1, z_2] \mid z_1 < z_2, z_1 \in Z' \cup \{-\infty\}, z_2 \in Z' \cup \{\infty\}\}$ where $Z' = \{z \mid min \leq z \leq max\}$
  - $\bot \in Var_* \to Interval, \bot(x) = \emptyset \quad \forall x \in Var_*$
  - $\top \in Var_* \to Interval, \top(x) = [-\infty, +\infty] \quad \forall x \in Var_*$
- $\mathcal{F} = \{f^I : L \to L \mid f^I \ is \ monotone\}$
- Analysis functions f^s are shown in Table 5.
- Flow graph: $F = flow(S_*)$
- Labels: $E = \{init(S_*)\}$
- Initial element: $\iota(x) = [-\infty, +\infty] \ for \ all \ variables \ x$

$Block^l$	$f_l^I(\widehat{\sigma})$
$[int\ x]^l$	$\begin{cases}\widehat{\sigma}[x \mapsto [bottom(0), top(0)]] & if\ \widehat{\sigma} \neq \bot \\ \bot & otherwise\end{cases}$
$[int[n]\ A]^l$	$\begin{cases}\widehat{\sigma}[A \mapsto [bottom(0), top(0)]] & if\ n > 0\ and\ \widehat{\sigma} \neq \bot \\ \bot & otherwise\end{cases}$
$[\{int\ fst;\ int\ snd\}R]^l$	$\begin{cases}\widehat{\sigma}[R.fst \mapsto [bottom(0), top(0)], R.snd \mapsto [bottom(0), top(0)]] & if\ \widehat{\sigma} \neq \bot \\ \bot & otherwise\end{cases}$
$[x := a]^l$	$\begin{cases}\widehat{\sigma}[x \mapsto A_I[\![a]\!]\widehat{\sigma}] & if\ \widehat{\sigma} \neq \bot \\ \bot & otherwise\end{cases}$
$[A[a] := a']^l$	$\begin{cases}\bot & if A_I[\![a]\!]\widehat{\sigma} \sqsubseteq [-\infty, bottom(-1)]\ or\ \widehat{\sigma} = \bot \\ \widehat{\sigma}[A \mapsto A_I[\![a']\!]\widehat{\sigma} \cup \widehat{\sigma}(A)] & otherwise\end{cases}$
$[R.fst := a]^l$	$\begin{cases}\widehat{\sigma}[R.fst \mapsto A_I[\![a]\!]\widehat{\sigma}] & if\ \widehat{\sigma} \neq \bot \\ \bot & otherwise\end{cases}$
$[R.snd := a]^l$	$\begin{cases}\widehat{\sigma}[R.snd \mapsto A_I[\![a]\!]\widehat{\sigma}] & if\ \widehat{\sigma} \neq \bot \\ \bot & otherwise\end{cases}$
$[R := (a_1, a_2)]^l$	$\begin{cases}\widehat{\sigma}[R.fst \mapsto A_I[\![a_1]\!]\widehat{\sigma}, R.snd \mapsto A_I[\![a_2]\!]\widehat{\sigma}] & if\ \widehat{\sigma} \neq \bot \\ \bot & otherwise\end{cases}$
$[read\ x]^l$	$\begin{cases}\widehat{\sigma}[x \mapsto [-\infty, +\infty]] & if\ \widehat{\sigma} \neq \bot \\ \bot & otherwise\end{cases}$
$[read\ A[a]]^l$	$\begin{cases}\bot & if A_I[\![a]\!]\widehat{\sigma} \sqsubseteq [-\infty, bottom(-1)]\ or\ \widehat{\sigma} = \bot \\ \widehat{\sigma}[A \mapsto [-\infty, +\infty]] & otherwise\end{cases}$
$[read\ R.fst]^l$	$\begin{cases}\widehat{\sigma}[R.fst \mapsto [-\infty, +\infty]] & if\ \widehat{\sigma} \neq \bot \\ \bot & otherwise\end{cases}$
$[read\ R.snd]^l$	$\begin{cases}\widehat{\sigma}[R.snd \mapsto [-\infty, +\infty]] & if\ \widehat{\sigma} \neq \bot \\ \bot & otherwise\end{cases}$
$[write\ a]^l$	$\widehat{\sigma}$
$[write\ A[a]]^l$	$\begin{cases}\bot & if A_I[\![a]\!]\widehat{\sigma} \sqsubseteq [-\infty, bottom(-1)]\ or\ \widehat{\sigma} = \bot \\ \widehat{\sigma} & otherwise\end{cases}$
$[b]^l$	$\widehat{\sigma}$

Table 5: Transfer functions for MICRO-C language

Since Z' and the $Interval$ is determined by min and max, we decided to use functions $bottom(x)$ and $top(x)$. $bottom(x)$ is the largest integer that is in the interval $[min, max] \cup -\infty$ and smaller or equal than x. $top(x)$ is the smallest integer that is in the interval $[min, max] \cup \infty$ and greater or equal than x.

$A_I : AExp \to (\widehat{State_I} \to Interval)$ determines the signs of arithmetic expressions:

$$A_I[\![n]\!]\widehat{\sigma} = \begin{cases}\bot & if\ \widehat{\sigma} = \bot \\ [n, n] & if\ min \leq n \leq max\ and\ \widehat{\sigma} \neq \bot \\ [-\infty, min] & if\ n < min\ and\ \widehat{\sigma} \neq \bot \\ [max, \infty] & if\ n > max\ and\ \widehat{\sigma} \neq \bot\end{cases}$$

$$A_I[\![x]\!]\widehat{\sigma} = \widehat{\sigma}(x)$$

$$A_I[\![A[a]]\!]\widehat{\sigma} = \begin{cases} \bot & if A_I[\![a]\!]\widehat{\sigma} \sqsubseteq [-\infty, bottom(-1)] \\ \widehat{\sigma}(A) & otherwise \end{cases}$$
$$A_I[\![R.fst]\!]\widehat{\sigma} = \widehat{\sigma}(R.fst)$$
$$A_I[\![R.snd]\!]\widehat{\sigma} = \widehat{\sigma}(R.snd)$$
$$A_I[\![a_1 \ \widehat{op_a} \ a_2]\!]\widehat{\sigma} = A_I[\![a_1]\!]\widehat{\sigma} \ \widehat{op_a} \ A_I[\![a_2]\!]\widehat{\sigma}$$

$\widehat{\mathbf{op_a}} = +$

$interval_1 + interval_2 = [k_1, k_2]$ *where* $interval_1 = [k_{11}, k_{12}]$ *and* $interval_2 = [k_{21}, k_{22}]$

$$k_1 = \begin{cases} \bot & if\ interval_1 = \bot\ or\ interval_2 = \bot \\ k_{11} + k_{21} & if\ min \leq k_{11} + k_{21} \leq max \\ -\infty & if\ k_{11} + k_{21} < min \\ max & if\ k_{11} + k_{21} > max \end{cases}$$

$$k_2 = \begin{cases} \bot & if\ interval_1 = \bot\ or\ interval_2 = \bot \\ k_{12} + k_{22} & if\ min \leq k_{12} + k_{22} \leq max \\ \infty & if\ k_{12} + k_{22} > max \\ min & if\ k_{12} + k_{22} < min \end{cases}$$

$\widehat{\mathbf{op_a}} = -$

$interval_1 - interval_2 = [k_1, k_2]$ *where* $interval_1 = [k_{11}, k_{12}]$ *and* $interval_2 = [k_{21}, k_{22}]$

$$k_1 = \begin{cases} \bot & if\ interval_1 = \bot\ or\ interval_2 = \bot \\ k_{11} - k_{21} & if\ min \leq k_{11} - k_{21} \leq max \\ -\infty & if\ k_{11} - k_{21} < min \\ max & if\ k_{11} - k_{21} > max \end{cases}$$

$$k_2 = \begin{cases} \bot & if\ interval_1 = \bot\ or\ interval_2 = \bot \\ k_{12} - k_{22} & if\ min \leq k_{12} - k_{22} \leq max \\ \infty & if\ k_{12} - k_{22} > max \\ min & if\ k_{12} - k_{22} < min \end{cases}$$

$\widehat{\mathbf{op_a}} = *$

$interval_1 * interval_2 = [k_1, k_2]$ *where* $interval_1 = [k_{11}, k_{12}]$ *and* $interval_2 = [k_{21}, k_{22}]$

Let $[z_1, z_2] = [min(k_{11}{*}k_{21}, k_{11}{*}k_{22}, k_{12}{*}k_{21}, k_{12}{*}k_{22}), max(k_{11}{*}k_{21}, k_{11}{*}k_{22}, k_{12}{*}k_{21}, k_{12}{*}k_{22})]$

$$k_1 = \begin{cases} \bot & if\ interval_1 = \bot\ or\ interval_2 = \bot \\ z_1 & if\ min \leq z_1 \leq max \\ -\infty & if\ z_1 < min \\ max & if\ z_1 > max \end{cases}$$

$$k_2 = \begin{cases} \bot & if\ interval_1 = \bot\ or\ interval_2 = \bot \\ z_2 & if\ min \leq z_2 \leq max \\ \infty & if\ z_2 > max \\ min & if\ z_2 < min \end{cases}$$

$\widehat{\mathbf{op_a}} = /$
$interval_1/interval_2 = [k_1, k_2]$ *where* $interval_1 = [k_{11}, k_{12}]$ *and* $interval_2 = [k_{21}, k_{22}]$
Let $[z_1, z_2] = [min(\lfloor k_{11}/k_{21} \rfloor, \lfloor k_{11}/k_{22} \rfloor, \lfloor k_{12}/k_{21} \rfloor, \lfloor k_{12}/k_{22} \rfloor),$
$max(\lfloor k_{11}/k_{21} \rfloor, \lfloor k_{11}/k_{22} \rfloor, \lfloor k_{12}/k_{21} \rfloor, \lfloor k_{12}/k_{22} \rfloor)]$

$$k_1 = \begin{cases} \bot & if\ interval_1 = \bot\ \ or\ interval_2 = \bot\ \ or\ interval_2 = [0,0] \\ z_1 & if\ min \leq z_1 \leq max \\ -\infty & if\ z_1 < min \\ max & if\ z_1 > max \end{cases}$$

$$k_2 = \begin{cases} \bot & if\ interval_1 = \bot\ \ or\ interval_2 = \bot\ \ or\ interval_2 = [0,0] \\ z_2 & if\ min \leq z_2 \leq max \\ \infty & if\ z_2 > max \\ min & if\ z_2 < min \end{cases}$$

2.6.3 Example Program

We will visit the example program in Section 2.5.3.

Initial element: $\iota = [x \mapsto [-\infty, +\infty], y \mapsto [-\infty, +\infty], a \mapsto [-\infty, +\infty]]$

Set of interval points: $\{-2, -1, 0, 1, 2\}$

Equations:

$$\begin{aligned} A_\bullet(1) &= f^I_{int\ x}(\iota) \\ A_\bullet(2) &= f^I_{int\ y}(A_\bullet(1)) \\ A_\bullet(3) &= f^I_{int[10]\ a}(A_\bullet(2)) \\ A_\bullet(4) &= f^I_{x:=3}(A_\bullet(3)) \\ A_\bullet(5) &= f^I_{y:=-2}(A_\bullet(4)) \\ A_\bullet(6) &= f^I_{x>2}(A_\bullet(5)) \\ A_\bullet(7) &= f^I_{x:=x-1}(A_\bullet(6)) \\ A_\bullet(8) &= f^I_{x:=x+1}(A_\bullet(6)) \\ A_\bullet(9) &= f^I_{a[x]:=y*y}(A_\bullet(7) \cup A_\bullet(8)) \end{aligned}$$

Solution:

$A_\bullet(.)$	x	y	a
1	[0,0]	$[-\infty, +\infty]$	$[-\infty, +\infty]$
2	[0,0]	[0,0]	$[-\infty, +\infty]$
3	[0,0]	[0,0]	[0,0]
4	$[2, +\infty]$	[0,0]	[0,0]
5	$[2, +\infty]$	[-2,-2]	[0,0]
6	$[2, +\infty]$	[-2,-2]	[0,0]
7	$[1, +\infty]$	[-2,-2]	[0,0]
8	$[2, +\infty]$	[-2,-2]	[0,0]
9	$[1, +\infty]$	[-2,-2]	$[-\infty, 0]$

2.6.4 Improvements

As it can be seen from the transfer functions on Table 5, we haven't dealt with boolean expressions. One can use the result of the tests to pass different information of the branches of the boolean expressions.

To improve the analysis we can change analysis function for boolean expressions: changing labeling $[b]^l$ to $[b]^{l_t,l_f}$, letting f_{l_t} be the analysis function for the true branch and f_{l_f} be the analysis function for the false branch.

Two analysis functions for $[b]^{l_t,l_f}$ then would be:

$$f_{l_t}(\widehat{\sigma}) = \sqcup\{\widehat{\sigma'} \in Basic(\widehat{\sigma}) \mid tt \in \mathcal{B}_I[\![b]\!]\widehat{\sigma'}\}$$
$$f_{l_f}(\widehat{\sigma}) = \sqcup\{\widehat{\sigma'} \in Basic(\widehat{\sigma}) \mid ff \in \mathcal{B}_I[\![b]\!]\widehat{\sigma'}\}$$

As an example for the interval $[-2, 2]$, $Basic(\widehat{\sigma}) = \{[-2,-2], [-1,-1], [0,0], [1,1], [2,2]\}$. Another example for out of range intervals like $[2, \infty]$, $Basic(\widehat{\sigma}) = \{[2, \infty]\}$

$\mathcal{B}_S : BExp \to (\widehat{State_I} \to \mathcal{P}(\{tt, ff\}))$ determine the signs of boolean expressions:

$$\mathcal{B}_I[\![a_1\ op_r\ a_2]\!]\widehat{\sigma'} = A_S[\![a_1]\!]\widehat{\sigma'}\ \widehat{op_r}\ A_S[\![a_2]\!]\widehat{\sigma'}$$
$$\mathcal{B}_I[\![b_1\ op_b\ b_2]\!]\widehat{\sigma'} = \mathcal{B}_S[\![b_1]\!]\widehat{\sigma'}\ \widehat{op_b}\ \mathcal{B}_S[\![b_2]\!]\widehat{\sigma'}$$
$$\mathcal{B}_I[\![\neg b]\!]\widehat{\sigma'} = \{\neg t \mid t \in \mathcal{B}_s[\![b]\!]\widehat{\sigma'}\}$$

An example $\widehat{op_r} = <$:
$interval_1 = [k_1^-, k_1^+],\ interval_2 = [k_2^-, k_2^+]$

$$interval_1\ <\ interval_2 = \begin{cases} \{\} & if\ interval_1 = \bot\ \ or\ interval_2 = \bot \\ \{tt\} & if\ k_1^+ < k_2^- \\ \{ff\} & if\ k_1^- \geq k_2^+ \\ \{tt, ff\} & otherwise \end{cases}$$

2.7 Extensions

Extensions discussed in 1.2.5, *continue* and *break* statements either return $\emptyset$ or $\widehat{\sigma}$ for all transfer functions of our analyses.

For Reaching Definition Analysis, Live Variables Analysis, Faint Variables Analysis and Dangerous Variables Analysis *kill* and *gen* functions for both *break* and *continue* are equal to $\emptyset$.

For Detection of Signs Analysis and Interval Analysis, transfer functions are equal to $\widehat{\sigma}$.

3 Implementation

3.1 Implementation Choices

3.1.1 Abstract Syntax Tree

Abstract syntax tree is chosen to be represented as a recursive data structure which will be implemented in Java as class objects.

A MICRO-C program will be represented as an object. For a more concrete representation, Figures 8, 9, 10, 11 are provided.

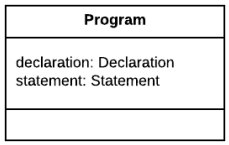


Figure 8: Program class

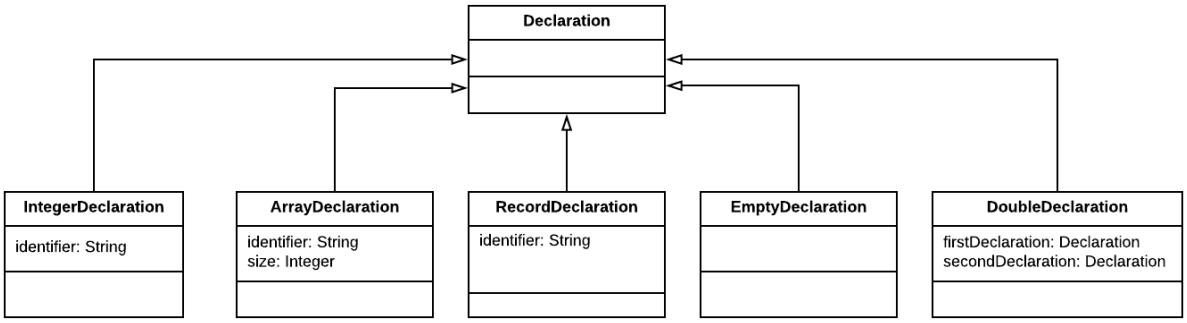


Figure 9: Declaration class

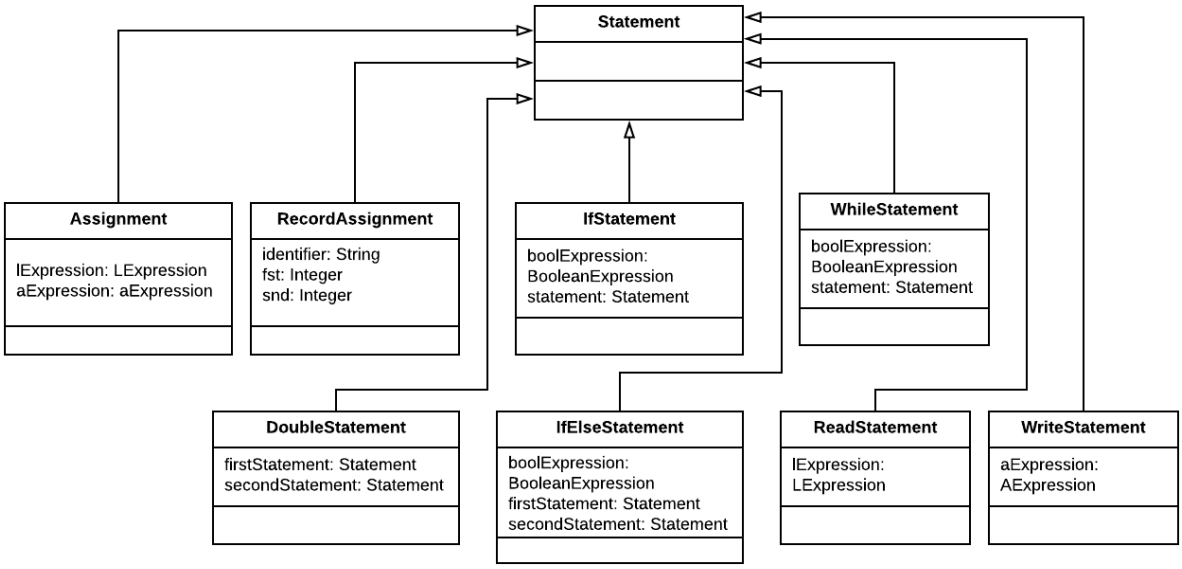


Figure 10: Statement class

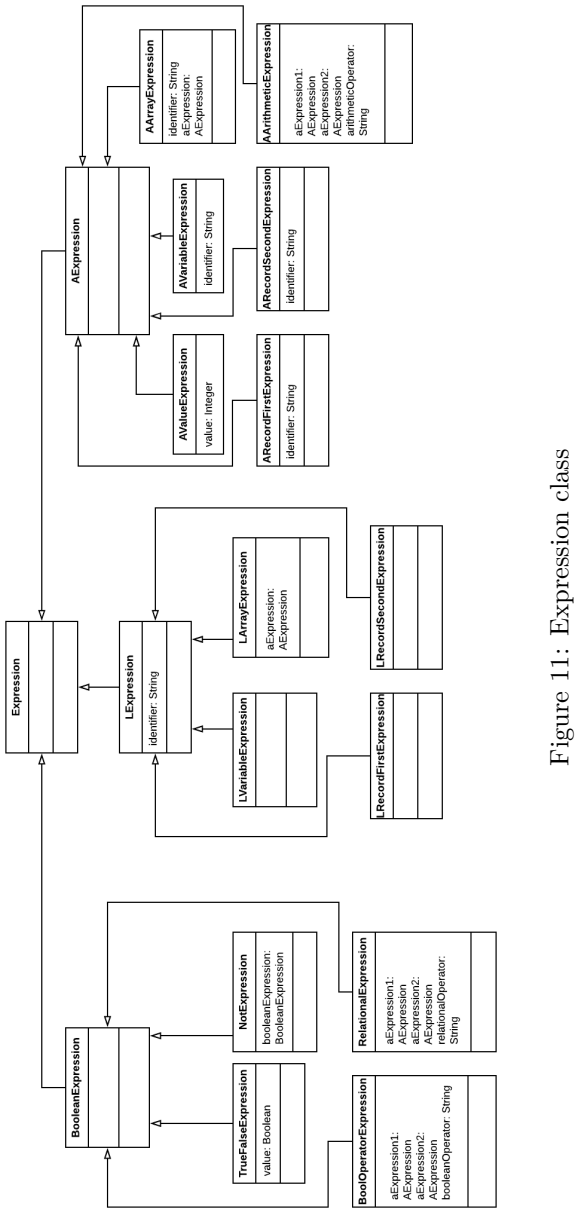


Figure 11: Expression class

Below example MICRO-C program in Section 2.1.3 is encoded.

program: *Program*
  declaration: *DoubleDeclaration*
    firstDeclaration: *DoubleDeclaration*
      firstDeclaration: *IntegerDeclaration*
        identifier: *String* = "x"
      secondDeclaration: *ArrayDeclaration*
        identifier: *String* = "a"
        size: *Integer* = 10
    secondDeclaration: *RecordDeclaration*
      identifier: *String* = "r"

statement: *DoubleStatement*
  firstStatement: *DoubleStatement*
    firstStatement: *DoubleStatement*
      firstStatement: *WhileStatement*
        boolExpression: *RelationalExpression*
          aExpression1: *AVariableExpression*
            identifier: *String* = "x"
          aExpression2: *AValueExpression*
            value: *Integer* = 0
          relationalOperator: *String* = ">"
        statement: *IfElseStatement*
          boolExpression: *RelationalExpression*
            aExpression1: *AVariableExpression*
              identifier: *String* = "x"
            aExpression2: *AValueExpression*
              value: *Integer* = 2
            relationalOperator: *String* = "=="
          firstStatement: *Assignment*
            lExpression: *LVariableExpression*
              identifier: *String* = "x"
            aExpression: *AArithmeticExpression*
              aExpression1: *AVariableExpression*
                identifier: *String* = "x"
              aExpression2: *AValueExpression*
                value: *Integer* = 1
              arithmeticOperator: *String* = "-"
          secondStatement: *Assignment*
            lExpression: *LVariableExpression*
              identifier: *String* = "x"
            aExpression: *AArithmeticExpression*
              aExpression1: *AVariableExpression*
                identifier: *String* = "x"
              aExpression2: *AValueExpression*
                value: *Integer* = 2
              arithmeticOperator: *String* = "-"
      secondStatement: *Assignment*
        lExpression: *LVariableExpression*
          identifier: *String* = "x"
        aExpression: *AValueExpression*
          value: *Integer* = 1
    secondStatement: *Assignment*
      lExpression: *LArrayExpression*
        identifier: *String* = "a"
        aExpression: *AVariableExpression*
          identifier: *String* = "x"
      aExpression: *AValueExpression*

value: *Integer* = 3
secondStatement: *Assignment*
lExpression: *LRecordFirstExpression*
identifier: *String* = "r"
aExpression: *AArrayExpression*
identifier: *String* = "a"
aExpression: *AVariableExpression*
identifier: *String* = "x"

3.1.2 Program Graph

We have decided to use the program graph in our implementation. This decision is a preference considering one can easily construct a program graph with a given flow graph. We preferred to contain data in the edges rather than nodes so we have gone with the program graph.

Program graph is chosen to be represented as two classes: Node and Edge. Edges will contain necessary information about the program point i.e. start node, program step and end node. Nodes will contain label ids. Start node's label id will always be 0 whereas end node's will be -1. Class diagram for Node and Edge classes are provided in Figure 12.

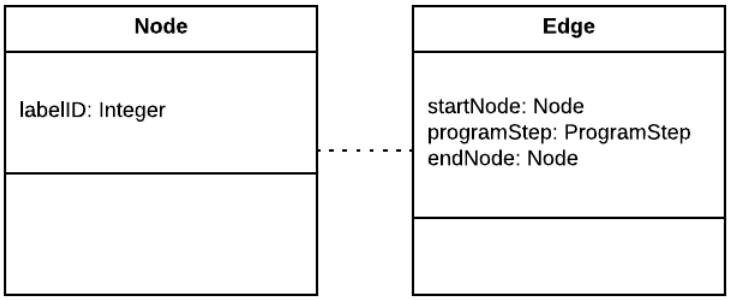


Figure 12: Node and Edge classes

3.1.3 Worklist

We have implemented a generic worklist algorithm following through the pseudo code in Lecture 8. Two containers we chose to implement are FIFO and LIFO algorithms. So we have two `insert` and one `extract` functions as follows:

```
1   public void insert(Integer constraint) {
2       if(type.equals("LIFO")) {
3           this.w.add(0, constraint);
4       }
5       else if(type.equals("FIFO")) {
6           this.w.add(constraint);
7       }
8   }
9
10  public Integer extract() {
11      Integer result = this.w.get(0);
12      this.w.remove(0);
```

```
        return result;
    }
```

Our `Worklist` class has three fields:

- `Analysis analysisType`: Different analyses (`ReachingDefinition` and `DetectionOfSigns` objects in our case)
- `ArrayList<Integer> w`: Worklist consisting of constraint labels
- `String type`: Either "FIFO" or "LIFO"

We wanted our worklist algorithm to be generic for different analyses so we decided to choose `ArrayList<ArrayList<String>>` for type of constraints. An example would be `[["x","0","1"], ["y","1","2"], ...]` for Reaching Definition Analysis and `[["x","+"], ["x","0"], ["y","-"], ["y","0"], ...]` for Detection of Signs Analysis.

Since there can be illegal indexing and division by zero in the Detection of Signs analysis, we needed to handle the case whether calculated constraint contains all variables. If not, we didn't update the `analysisSet`. Code snippet for this procedure including the main worklist algorithm is as follows:

```
public void solve() {
    while(!w.isEmpty()) {
        Integer currentIndex = extract();
        ArrayList<ArrayList<String>> newSol = analysisType.createConstraints(currentIndex);

        int flag2 = 0;
        for(String variable : analysisType.getCfa().variableList) {
            int flag = 0;
            for(ArrayList<String> arrayList : newSol) {
                if(arrayList.contains(variable)) {
                    flag = 1;
                    break;
                }
            }
            if(flag == 0) flag2 = 1;
        }
        if(flag2 == 1) continue;

        if(newSol.containsAll(analysisType.getAnalysisSet().get(currentIndex)) && newSol.
            ↪ size() > analysisType.getAnalysisSet().get(currentIndex).size()) {
            analysisType.updateAnalysisSet(currentIndex, newSol);
            for(Integer i : analysisType.getInfl().get(currentIndex)) {
                insert(i);
            }
        }
    }
}
```

3.1.4 Analyses

We have implemented two analyses: Reaching Definitions and Detection of Signs. For both classes, we have implemented an `initialize` method that goes through the flow graph, creates empty constraints (`analysisSet`) and fills the influence sets (`infl`). `analysisSet` is the type `ArrayList<ArrayList<String>>` as mentioned in Section 3.1.3.

For calculating constraints, we have implemented method `createConstraints` that takes a label. This method is unique to different analysis classes.

Reaching Definitions: If label is 0, which means that it is the initial constraint, implementation goes through the variables and returns `[[`*`variable name`*`, ?, 0], ...]`.

If label is not 0, then implementation goes through the edges and finds the nodes that connected to the given label. For each node it finds, it takes the analysis set (current state), calls `kill` and `gen` functions to create the constraint for the given label. Below is the code snippet for this procedure:

```
    for(Edge edge : cfa.edgeList) {
        if(label == edge.getEndNode().getLabelId()) {
            ArrayList<ArrayList<String>> temp =
                    (ArrayList<ArrayList<String>>) analysisSet.get(edge.getStartNode().
                        ↪ getLabelId()).clone();
            temp.removeAll(kill(edge, analysisSet.get(edge.getStartNode().getLabelId())));
            temp.addAll(gen(edge));
            for(ArrayList<String> arrayList : temp) {
                if(!result.contains(arrayList)) {
                    result.add(arrayList);
                }
            }
        }
    }
```

Detection of Signs: Again, if the label is 0, then implementation adds initial value to all variables (`[[`*`variable name`*`, -], [`*`variable name`*`, 0], [`*`variable name`*`, +]]`).
If the label is not 0, then implementation goes through the edges just like in Reaching Definitions. This time for each node it takes the analysis set (current state), calls `transfer function` to create constraint to the given label. Method `transferFunction` calls `signFunction` if necessary. Below is the code snippet for this procedure:

```
    for(Edge edge : cfa.edgeList) {
        if(label == edge.getEndNode().getLabelId()) {
            ArrayList<ArrayList<String>> temp = transferFunction(edge, analysisSet.get(edge.
                ↪ getStartNode().getLabelId()));
            for(ArrayList<String> arrayList : temp) {
                if(!result.contains(arrayList)) {
                    result.add(arrayList);
                }
            }
        }
    }
```

We also captured illegal array indexing and division by zero in our Detection of Signs implementation. The transfer function returns empty set, if there is an error, i.e. negative array size declaration, negative array indexing and division by zero. Handling empty sets in the worklist algorithm is mentioned in Section 3.1.3.

3.1.5 Example Executions

We have written two example MICRO-C programs to test our implementation. We applied Reaching Definitions and Detection of Signs Analysis on both examples.

Example 1

```
int[3] a;                          <l1>
int x;                             <l2>
x := 0;                            <l3>
while x>3 {                        <l4>
    a[x] := 1;                     <l5>
    if x==2 <l6> a[x] := 3;        <l7>
}
x := 5;                            <l8>
```

Reaching Definitions:

- *RD0* :[x, ?, 0] [a, ?, 0]
- *RD-1* :[a, 0, 1] [a, 7, 3] [a, 5, 6] [x, 4, -1]
- *RD1* :[x, ?, 0] [a, 0, 1]
- *RD2* :[a, 0, 1] [x, 1, 2]
- *RD3* :[a, 0, 1] [x, 2, 3] [a, 7, 3] [a, 5, 6]
- *RD4* :[a, 0, 1] [x, 2, 3] [a, 7, 3] [a, 5, 6]
- *RD5* :[a, 0, 1] [x, 2, 3] [a, 7, 3] [a, 5, 6]
- *RD6* :[x, 2, 3] [a, 5, 6]
- *RD7* :[x, 2, 3] [a, 5, 6]

Detection of Signs:

- *DS0* :[x, -] [x, 0] [x, +] [a, -] [a, 0] [a, +]
- *DS-1* :[a, 0] [a, +] [x, +]
- *DS1* :[x, -] [x, 0] [x, +] [a, 0]
- *DS2* :[a, 0] [x, 0]
- *DS3* :[a, 0] [x, 0] [a, +]
- *DS4* :[a, 0] [x, 0] [a, +]
- *DS5* :[a, 0] [x, 0] [a, +]
- *DS6* :[a, 0] [x, 0] [a, +]
- *DS7* :[a, 0] [x, 0] [a, +]

Example 2

```
1  int x;                                              <l1>
2  int[10] a;                                          <l2>
3  {int fst; int snd} r;                               <l3>
4  x := 10;                                            <l4>
5  while x>0 {                                         <l5>
6      if x==2 <l6> then x := x-1; <l7> a[2] := 3;     <l8>
7      else x := x-2;                                  <l9>
8  }
9  x := 1;                                             <l10>
10 a[x] := 3;                                          <l11>
11 read r.snd;                                         <l12>
12 r.fst := a[x] + r.snd;                              <l13>
13 write r.fst;                                        <l14>
```

Reaching Definitions:

- *RD0* :[x, ?, 0] [a, ?, 0] [r.fst, ?, 0] [r.snd, ?, 0]
- *RD-1* :[x, 5, 10] [a, 10, 11] [r.snd, 11, 12] [r.fst, 12, 13]
- *RD1* :[a, ?, 0] [r.fst, ?, 0] [r.snd, ?, 0] [x, 0, 1]

- *RD2* :[r.fst, ?, 0] [r.snd, ?, 0] [x, 0, 1] [a, 1, 2]
- *RD3* :[x, 0, 1] [a, 1, 2] [r.fst, 2, 3] [r.snd, 2, 3]
- *RD4* :[a, 1, 2] [r.fst, 2, 3] [r.snd, 2, 3] [x, 3, 4] [a, 9, 4] [x, 8, 4] [x, 7, 9]
- *RD5* :[a, 1, 2] [r.fst, 2, 3] [r.snd, 2, 3] [x, 3, 4] [a, 9, 4] [x, 8, 4] [x, 7, 9]
- *RD6* :[a, 1, 2] [r.fst, 2, 3] [r.snd, 2, 3] [x, 3, 4] [a, 9, 4] [x, 8, 4] [x, 7, 9]
- *RD7* :[a, 1, 2] [r.fst, 2, 3] [r.snd, 2, 3] [x, 3, 4] [a, 9, 4] [x, 8, 4] [x, 7, 9]
- *RD8* :[a, 1, 2] [r.fst, 2, 3] [r.snd, 2, 3] [x, 3, 4] [a, 9, 4] [x, 8, 4] [x, 7, 9]
- *RD9* :[a, 1, 2] [r.fst, 2, 3] [r.snd, 2, 3] [a, 9, 4] [x, 7, 9]
- *RD10* :[a, 1, 2] [r.fst, 2, 3] [r.snd, 2, 3] [a, 9, 4] [x, 5, 10]
- *RD11* :[r.fst, 2, 3] [r.snd, 2, 3] [x, 5, 10] [a, 10, 11]
- *RD12* :[r.fst, 2, 3] [x, 5, 10] [a, 10, 11] [r.snd, 11, 12]
- *RD13* :[x, 5, 10] [a, 10, 11] [r.snd, 11, 12] [r.fst, 12, 13]

Detection of Signs:

- *DS0* :[x, -] [x, 0] [x, +] [a, -] [a, 0] [a, +] [r.fst, -] [r.fst, 0] [r.fst, +] [r.snd, -] [r.snd, 0] [r.snd, +]
- *DS-1* :[a, 0] [a, +] [x, +] [r.snd, -] [r.snd, 0] [r.snd, +] [r.fst, -] [r.fst, 0] [r.fst, +]
- *DS1* :[a, -] [a, 0] [a, +] [r.fst, -] [r.fst, 0] [r.fst, +] [r.snd, -] [r.snd, 0] [r.snd, +] [x, 0]
- *DS2* :[r.fst, -] [r.fst, 0] [r.fst, +] [r.snd, -] [r.snd, 0] [r.snd, +] [x, 0] [a, 0]
- *DS3* :[x, 0] [a, 0] [r.fst, 0] [r.snd, 0]
- *DS4* :[a, 0] [r.fst, 0] [r.snd, 0] [x, +] [a, +] [x, -] [x, 0]
- *DS5* :[a, 0] [r.fst, 0] [r.snd, 0] [x, +] [a, +] [x, -] [x, 0]
- *DS6* :[a, 0] [r.fst, 0] [r.snd, 0] [x, +] [a, +] [x, -] [x, 0]
- *DS7* :[a, 0] [r.fst, 0] [r.snd, 0] [x, +] [a, +] [x, -] [x, 0]
- *DS8* :[a, 0] [r.fst, 0] [r.snd, 0] [x, +] [a, +] [x, -] [x, 0]
- *DS9* :[a, 0] [r.fst, 0] [r.snd, 0] [a, +] [x, -] [x, 0] [x, +]
- *DS10* :[a, 0] [r.fst, 0] [r.snd, 0] [a, +] [x, +]
- *DS11* :[a, 0] [r.fst, 0] [r.snd, 0] [a, +] [x, +]
- *DS12* :[a, 0] [r.fst, 0] [a, +] [x, +] [r.snd, -] [r.snd, 0] [r.snd, +]
- *DS13* :[a, 0] [a, +] [x, +] [r.snd, -] [r.snd, 0] [r.snd, +] [r.fst, -] [r.fst, 0] [r.fst, +]

3.1.6 Different Versions of Example Executions

In order to test our implementation, we decided to do three modifications to our Example 2.

Example 2-1

We have changed line 2 in Example 2 for testing Detection of Signs when array is declared with a negative index:

```
1 int x;                                          <l1>
2 int[-10] a;                                     <l2>
3 ...
```

As expected, we got the following result for Detection of Signs Analysis:

- *DS0* :[x, -] [x, 0] [x, +] [a, -] [a, 0] [a, +] [r.fst, -] [r.fst, 0] [r.fst, +] [r.snd, -] [r.snd, 0] [r.snd, +]
- *DS-1* :
- *DS1* :[a, -] [a, 0] [a, +] [r.fst, -] [r.fst, 0] [r.fst, +] [r.snd, -] [r.snd, 0] [r.snd, +] [x, 0]
- *DS2* :
- *DS3* :
- *DS4* :
- *DS5* :
- *DS6* :
- *DS7* :
- *DS8* :
- *DS9* :
- *DS10* :
- *DS11* :
- *DS12* :
- *DS13* :

Example 2-2

We have changed line 6 in Example 2 for testing Detection of Signs when array is reached with a negative index:

```
1 int x;                                          <l1>
2 int[10] a;                                      <l2>
3 {int fst; int snd} r;                           <l3>
4 x := 10;                                        <l4>
5 while x>0 {                                     <l5>
6     if x==2 <l6> then x := x-1; <l7> a[-2] := 3; <l8>
7     else x := x-2;                              <l9>
8 }
9 ...
```

As expected, we got the following result for Detection of Signs Analysis:

- *DS0* :[x, -] [x, 0] [x, +] [a, -] [a, 0] [a, +] [r.fst, -] [r.fst, 0] [r.fst, +] [r.snd, -] [r.snd, 0] [r.snd, +]
- *DS-1* :[a, 0] [x, +] [a, +] [r.snd, -] [r.snd, 0] [r.snd, +] [r.fst, -] [r.fst, 0] [r.fst, +]
- *DS1* :[a, -] [a, 0] [a, +] [r.fst, -] [r.fst, 0] [r.fst, +] [r.snd, -] [r.snd, 0] [r.snd, +] [x, 0]
- *DS2* :[r.fst, -] [r.fst, 0] [r.fst, +] [r.snd, -] [r.snd, 0] [r.snd, +] [x, 0] [a, 0]
- *DS3* :[x, 0] [a, 0] [r.fst, 0] [r.snd, 0]
- *DS4* :[a, 0] [r.fst, 0] [r.snd, 0] [x, +] [x, -] [x, 0]
- *DS5* :[a, 0] [r.fst, 0] [r.snd, 0] [x, +] [x, -] [x, 0]
- *DS6* :[a, 0] [r.fst, 0] [r.snd, 0] [x, +] [x, -] [x, 0]
- *DS7* :[a, 0] [r.fst, 0] [r.snd, 0] [x, +] [x, -] [x, 0]
- *DS8* :[a, 0] [r.fst, 0] [r.snd, 0] [x, +] [x, -] [x, 0]
- *DS9* :[a, 0] [r.fst, 0] [r.snd, 0] [x, -] [x, 0] [x, +]
- *DS10* :[a, 0] [r.fst, 0] [r.snd, 0] [x, +]
- *DS11* :[a, 0] [r.fst, 0] [r.snd, 0] [x, +] [a, +]
- *DS12* :[a, 0] [r.fst, 0] [x, +] [a, +] [r.snd, -] [r.snd, 0] [r.snd, +]
- *DS13* :[a, 0] [x, +] [a, +] [r.snd, -] [r.snd, 0] [r.snd, +] [r.fst, -] [r.fst, 0] [r.fst, +]

Example 2-3

We have changed line 9 in Example 2 for testing Detection of Signs when array is reached with a negative index:

```
1  int x;                                                    <l1>
2  int[10] a;                                                <l2>
3  {int fst; int snd} r;                                     <l3>
4  x := 10;                                                  <l4>
5  while x>0 {                                               <l5>
6      if x==2 <l6> then x := x-1; <l7> a[2] := 3;           <l8>
7      else x := x-2;                                        <l9>
8  }
9  x := -1;                                                  <l10>
10 a[x] := 3;                                                <l11>
11 ...
```

As expected, we got the following result for Detection of Signs Analysis:

- *DS0* :[x, -] [x, 0] [x, +] [a, -] [a, 0] [a, +] [r.fst, -] [r.fst, 0] [r.fst, +] [r.snd, -] [r.snd, 0] [r.snd, +]
- *DS-1* :
- *DS1* :[a, -] [a, 0] [a, +] [r.fst, -] [r.fst, 0] [r.fst, +] [r.snd, -] [r.snd, 0] [r.snd, +] [x, 0]
- *DS2* :[r.fst, -] [r.fst, 0] [r.fst, +] [r.snd, -] [r.snd, 0] [r.snd, +] [x, 0] [a, 0]
- *DS3* :[x, 0] [a, 0] [r.fst, 0] [r.snd, 0]
- *DS4* :[a, 0] [r.fst, 0] [r.snd, 0] [x, +] [a, +] [x, -] [x, 0]

- *DS5* :[a, 0] [r.fst, 0] [r.snd, 0] [x, +] [a, +] [x, -] [x, 0]
- *DS6* :[a, 0] [r.fst, 0] [r.snd, 0] [x, +] [a, +] [x, -] [x, 0]
- *DS7* :[a, 0] [r.fst, 0] [r.snd, 0] [x, +] [a, +] [x, -] [x, 0]
- *DS8* :[a, 0] [r.fst, 0] [r.snd, 0] [x, +] [a, +] [x, -] [x, 0]
- *DS9* :[a, 0] [r.fst, 0] [r.snd, 0] [a, +] [x, -] [x, 0] [x, +]
- *DS10* :[a, 0] [r.fst, 0] [r.snd, 0] [a, +] [x, -]
- *DS11* :
- *DS12* :
- *DS13* :