Computer Science Engineering

Final Report - 14 May 2022

Context Sensitive Constant Propagation

Ravindra kumar (CS21M050)

supervised by
Dr. V. Krishna Nandivada

Acknowledgements

We especially like to thank Dr. V Krishna Nandivada for his teachings and suggestions for the project topic. He also suggested some of the reading materials for the completion of this milestone.

Abstract

We are going to mention one of the classic problem in architecture independent code optimization which is Constant Propagation (CP). The Intermediate Representation (IR) of the program can have confluence points. We are going to extend the analysis and see the call site dependent constant propagation i.e. context sensitive constant propagation (CSCP).

We will be seeing the formal definition of CP and some of the classic algorithms in CP analysis. Few of them are from Kildall's algorithm, Wegmen et al.'s algorithm, Alfred Aho et al.'s algorithm for confluence point and some of the CP characteristics. Finally, we will see the work by David Callaham et al. on context sensitive interprocedural constant propagation (ICP).

Ullman et al. showed that CP problem is undecidable in nature. Therefore, all the above algorithms are sound and conservative in nature but none guarantee that they detect all constant in the IR.

In this paper, we have designed a new worklist based context sensitive CP. We show that it doesnot work for parallel program with an example. Therefore, We modifiy the algorithm into two phases (adding the edge phase and analysis phase) to support the parallel execution of program statements (like X10). We have provided many examples for the simplicity of understanding.

Keywords

Here your keywords: confluence point, call site.

Contents

0.1 Introduction and Motivation

Constant propagation (CP) is one of the early optimizations during architecture independent optimization. It is one of the well-known global flow analysis problem. Although constant folding is a different analysis but constant propagation usually implies constant propagation with constant folding.

Goal of CP analysis is to discover values that are constant on all execution paths and propagate these values as far as possible. Expressions whose operands are constants are evaluated and their values are propagated. All the algorithms mentioned in this report are conservative, therefore they are sound but incomplete as the problem is undecidable (proved by Ullman et al[1].).

CP helps in removing unreachable code (branch which is never executed). It helps to increase the performance of the program for example by evaluating the expressions inside loops at compile time and save many CPU cycles at runtime.

Allen et al. looked at a large sample of programs, and found that over 24 percent of all parameters to subroutines in PL/I are lexically constants. Presumably any global constant propagation algorithm would find additional ones. Moreover, in languages that make heavy use of generics or where runtime type information is needed, constant propagation has ever more potential to be helpful.

Many times functions are written with code reusablility in mind and most of their codes are either constants or without any parameters. Using CP we can evaluate it once and use its returned value every time the function call has same parameters.

It also has phase ordering relationship with many other optimization techniques like def-use analysis, Points-to analysis.

0.2 Related Work

The first algorithm mentioning the constant propagation analysis was from Killdall [2] in 1973. He gave the definition and one of the intraprocedural algorithmic solution. After that, many algorithms were proposed and some were proposed for a specific Intermediate representations or using some specific graph representations.
Ullman et al. [1] in 1977 showed that intraprocedural constant propagation is undecidable, with a simple example.
Most of the initial works were for intraprocedural analysis like Reif et al.'s algorithm , K. Kennedy's

algorithm, Wegmen et al.'s [3] algorithm. Reif et al.'s algorithm use a global value graph data structure which is based on p-graph. This algorithm is better than Kildall's [2] algorithm in terms of speed but has precision same as kildall's [2] algorithm i.e. the set of constants detected are equivalent in both algorithms. After almost seven years, Wegmen et al. [3] presented a paper which explains some of the older algorithms and a new algorithm which uses some of the features of all the older algorithms especially Wegbreit's algorithm. In their paper, they evaluate the branch conditions and they evaluate only the code portions which will be executed. Executing only portion of the whole code resulted in more constants detection in the programs with faster execution as some part of the code is not executed.
David Callaham et al. [4] in 1986 gave a collection of algorithm for interprocedural constant propagation. The presented algorithms in the paper had a trade off between the complexity of the algorithm vs the precision of the algorithm.

0.3 Definition and lattice

Definition 1. A variable $x \in$ **Var** has a constant value $c \in$ **Const** at a program point u if for every path reaching u along which a definition of x reaches u, the value of x is c (denoted by $x \leftarrow c$).

Some characteristic of CP framework analysis:

- CP is a forward data flow analysis.
- CP framework is monotone but not-distributive i.e. $f_3(f_1(m_0) \sqcap f_2(m_0)) < f_3(f_1(m_0)) \sqcap f_3(f_2(m_0))$. In simple words, the iterative solution Maximal Fixed Points (MFP) is safe but may be smaller than the Meet over all paths (MOP) solution. (From Alfread Aho et al. book. [5] Refer it for a simple proof.)

0.3.1 Lattice for Constant Propagation

Each element of CP Lattice is one of three:

- a lattice top element, $\top$: May be a constant(undetermined).
- a lattice bottom element, $\bot$: Constant value cannot be guaranteed.
- a constant value, c.

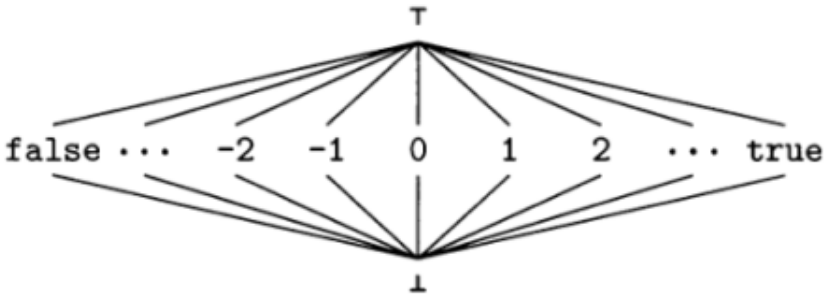

Figure 1: Constant Propagation Lattice

Rules for Meet ($\sqcap$):

- $any \sqcap \top = any$
- $any \sqcap \bot = \bot$
- $c_i \sqcap c_j = c_i$, if $c_i = c_j$
- $c_i \sqcap c_j = \bot$, if $c_i \neq c_j$

0.4 Algorithms

All the algorithms presented are forward analysis and monotone. They all terminate for a finite program, as each variable can be assigned twice a lattice value.

0.4.1 Flow function

Considering each statement as basic block (BB). At each program point, we need to maintain a map(M), consisting of variables mapping to a value, $m \in M, v \leftarrow m(v)$ where $m(v)$ is a value from lattice.
Let $m, m' \in M$

Definition 2. Flow function F: set of functions. f_s is flow function for statement $s : m' = f_s(m)$.

Algorithm to compute flow functions for a statement:

Algorithm 1 Flow function

Start with the entry node
for every statement s **do**
if s is not an assignment statement, then f_s is simply the identity function: $m' = m$.

If s is an assignment statement to variable v then $m' = f_s(m)$, where:

- $\forall v' \neq v, m'(v') = m(v')$
- if RHS of the statement is constant c, then $m'(v) = c$
- if RHS of the statement is a variable x, then $m'(v) = m(x)$
- if RHS is an expression (y op z),

$$m'(v) = \begin{cases} \texttt{m(y) op m(z),} & \texttt{if m(y) and m(z) are constant values,} \\ \bot, & \texttt{if either m(y) or m(z) is } \bot, \\ \top, & \texttt{otherwise} \end{cases}$$

- if the RHS is an expression that cannot be evaluated (e.g. a function call or assignment through a pointer) then, $m'(v) = \bot$

end for
At the merge point, get a meet of the flow maps (MOP-meet over all paths).

0.4.2 Killdall's Simple Constant algorithm

We initialize the Map to Top. We will use the program control flow graph for propagation of values. Till M change in two consecutive iterations , we will keep iterating. We will maintain a worklist which will have all the blocks in the program and marked as unvisited. For all the the unvisited successors of block currently in consideration, we will determine the values of maps using flow function and add it to worklist.

Algorithm 2 Simple Constant

$M[entry] = init$
do
$change = false$
$visited(b) = false, \forall statements(b)$ ▷ not visit the same Basic Block twice in one iteration. Helps in streamlining the time complexity of the analysis.
$worklist \leftarrow$ all statements
while worklist not empty **do**
$b = worklist.remove$ ▷ Note: b is a basic block.
$visited(b) = true$
$m' = f_b(m)$ ▷ As defined in the flow function subsection.
if visited(b') **then**
continue
else
$m[b']\sqcap = m'$
if m[b'] changes **then**
$worklist.add(b')$
$change = true$
end if
end if
end while
while (change is true)

0.4.3 Complexity

N is the number of assignment statements plus the number of expressions whose value is branched on in the program.
E is the number of edges in the program flow graph. If we are analysing on 3-address code then $\mathbf{E} = \mathbf{2} * \mathbf{N}$.
V is the number of variables in the program.

Since the lattice value of each variable can only be lowered twice, each node may be visited at most $\mathbf{2} * \mathbf{V} * \mathbf{I}$ times, where **I** is the number of in-edges into that node. Thus, the time required for Kildall's algorithm is $\mathbf{O}(\mathbf{E} * \mathbf{V})$ node visits and **V** operations during each node visit. Thus in worst-case running time is $\mathbf{O}(\mathbf{E} * \mathbf{V^2})$.

0.4.4 Interprocedural Constant Propagation

We are going to discuss two ways of Interprocedural CP.

Method Inlining

It is the most obvious solution. Converting the program into one large procedure program by inlining all the function calls at their call site. Recursion procedures can be handled in this scheme by introducing an explicit stack, combined with unrolling. It makes the analysis more precise by making the parameter aliasing explicit in the program.

In spite of its effectiveness and simplicity, it has server drawbacks. First, it increases the program's code size exponentially. Since CP performance in a single procedure is roughly linearly dependent on program size. Therefore, the overhead of inlining is exponential. Secondly, systematic use of inline expansion makes the program difficult to maintain. After each change to a single procedure, the system must either re-expand the program or attempt to recreate the change in the expanded, optimized program, a process that is likely to be very expensive. In spite of the drawbacks, inlining provide us with a standard by which to measure the effectiveness of other techniques.

Worklist approach

We are going to see Flow sensitive context sensitive interprocedural constant propagation. Whenever adding a procedure to the worklist, store the callsite info for that procedure too. We can assume call site information for root method of callgraph to be 0. We represent $p_c \in W$ as procedure p with callsite information c in the current working list W. We represent $S_{p,c} \in S^T$, as summary of method p at call site c. S^T is total summary list maintained during Interprocedural CP analysis.

We are assuming each method statements start with a unique line number. We could also have used the global line numbering as call cite information but we are going with the method-wise line numbering.

Summary of a procedure will contain lattice values for parameters, global variables and return variable. Insert a method p_c with call site information c to the worklist W only if it is not present in the worklist. Note that object declaration and definition statements are treated as cannot be evaluated statements, and M **contain only primitive datatype variables**.

Algorithm 3 Flow Sensitive Context Sensitive Interprocedural Constant Propagation

Require: Callgraph CG
  Initialize S^T variables with $\top$
  worklist = $CG.root_0$
  while worklist is not empty **do**
    $p_c = worklist.dequeue$
    $v_{old} \leftarrow$ return value from $S_{p,c}$
    Initialize the M map for the current method p_c as follows.
    Initialize first statement's input map $m \in M$ variables with lattice value from summary of p_c. If a variable is not present in summary then initialize it with $\top$. Initialize the output map m' with *top*.
    Initialize all the other statement's input map m and output map m' with $\top$.
    do
      $change = false$
      $statementWorklist \leftarrow$ all statements
      while statementWorklist not empty **do**
        $s = statementWorklist.remove$
        m and m' are the input and output map for statement s. Note $m, m' \in M$.
        $m = m'(t), \forall t \in pred(s)$
        if stmt s is not of the form x= y.foo(...) **then**
          $m' = m' \sqcap f_s(m)$ ▷ As defined in the flow function subsection and apply $m' \sqcap f_s(m)$ elementwise
          if m' changed **then**
            $statementWorklist.add(succ(s))$
            $change = true$
          end if
        else
          compute the actual arguments of s using M
          for each function q that may be called at s **do**
            Compute meet: current values of the formals of q (Using summary) and the actuals
            if the values of the arguments of q have changed **then**
              remember the meet value and add the $q_{\text{cite info s}}$ to the worklist.
            end if
            "Update" the value of $m'(x)$, as per the $S_{q,\text{cite info s}}$
          end for
          if $m'(x)$ changed **then**
            $statementWorklist.add(succ(s))$
            $change = true$
          end if
        end if
      end while
    while (change is true)
    v = compute the meet of all the return values of p_c s
    if $v \neq v_{old}$ **then**
      Using the call site information (c) of p_c add the caller of p_c to the worklist.
    end if
    Set the return value of p_c to v
  end while

To make the algorithm flow insensitive context sensitive interprocedural constant propagation. Previously while analysing a procedure we were defining a set of maps M for each statement ($m_s and m'_s$) in the procedure, now we will maintain a single map M which will be common for all the statements in the procedure.

Using the explanation given for time complexity for kildall's et al. algorithm and extending it to the

interprocedural analysis. As the number of function calls is constant as each parameter,global variable and return value can atmost be set 2 times. Therefore the number of times a function can be called is constant and therefore the above algorithm will terminate.

0.5 Example

```
/*
* Program: report.java
* Functionality: Example to show the soundness of the proposed Interprocedural Constant
    Propagation.
* Author: Ravindra kumar
* Last modifier: 25-April-2022
* Bugs: None
 */

class report{
    public static void main(String []args){
        X x = new X();
        int a = 10;
        int b = 20;
        int c;
        if(a<b)
            c = x.foo(a,b);
        else
            c = 30;
        int d = a+c;
        int e = x.foo(d,b);
    }
}
class X{
    int foo(int p,int q){
        return p;
    }
}

class Y extends X{
    int foo(int p, int q){
        return q;
```

```
32      }
33  }
```

Currently method Analysed	Current Worklist (Before analysis)	Summary Map (After analysis)	updated Worklist (After analysis)
Initialization	$\{\}$	$\{S_{main,0} = \{\}, S_{X:foo,main:16} = \{\top,\top,\top\}, S_{Y:foo,main:16} = \{\top,\top,\top\}, S_{X:foo,main:20} = \{\top,\top,\top\}, S_{Y:foo,main:20} = \{\top,\top,\top\}\}$	$\{Main_0\}$
$Main_0$	$\{\}$	$\{S_{main,0} = \{\}, S_{X:foo,main:16} = \{10,20,\top\}, S_{Y:foo,main:16} = \{10,20,\top\}, S_{X:foo,main:20} = \{40,20,\top\}, S_{Y:foo,main:20} = \{40,20,\top\}\}$	$\{X : foo_{main:16}, Y : foo_{main:16}, X : foo_{main:20}, Y : foo_{main:20}\}$
$X : foo_{main:16}$	$\{Y : foo_{main:16}, X : foo_{main:20}, Y : foo_{main:20}\}$	$\{S_{main,0} = \{\}, S_{X:foo,main:16} = \{10,20,10\}, S_{Y:foo,main:16} = \{10,20,\top\}, S_{X:foo,main:20} = \{40,20,\top\}, S_{Y:foo,main:20} = \{40,20,\top\}\}$	$\{Y : foo_{main:16}, X : foo_{main:20}, Y : foo_{main:20}, Main_0\}$
$Y : foo_{main:16}$	$\{X : foo_{main:20}, Y : foo_{main:20}, Main_0\}$	$\{S_{main,0} = \{\}, S_{X:foo,main:16} = \{10,20,10\}, S_{Y:foo,main:16} = \{10,20,20\}, S_{X:foo,main:20} = \{40,20,\top\}, S_{Y:foo,main:20} = \{40,20,\top\}\}$	$\{X : foo_{main:20}, Y : foo_{main:20}, Main_0\}$
$X : foo_{main:20}$	$\{Y : foo_{main:20}, Main_0\}$	$\{S_{main,0} = \{\}, S_{X:foo,main:16} = \{10,20,10\}, S_{Y:foo,main:16} = \{10,20,20\}, S_{X:foo,main:20} = \{40,20,40\}, S_{Y:foo,main:20} = \{40,20,\top\}\}$	$\{Y : foo_{main:20}, Main_0\}$
$Y : foo_{main:20}$	$\{Main_0\}$	$\{S_{main,0} = \{\}, S_{X:foo,main:16} = \{10,20,10\}, S_{Y:foo,main:16} = \{10,20,20\}, S_{X:foo,main:20} = \{40,20,40\}, S_{Y:foo,main:20} = \{40,20,20\}\}$	$\{Main_0\}$
$Main_0$	$\{\}$	$\{S_{main,0} = \{\}, S_{X:foo,main:16} = \{10,20,10\}, S_{Y:foo,main:16} = \{10,20,20\}, S_{X:foo,main:20} = \{\bot,20,40\}, S_{Y:foo,main:20} = \{\bot,20,20\}\}$	$\{X : foo_{main:20}, Y : foo_{main:20}\}$
$X : foo_{main:20}$	$\{Y : foo_{main:20}\}$	$\{S_{main,0} = \{\}, S_{X:foo,main:16} = \{10,20,10\}, S_{Y:foo,main:16} = \{10,20,20\}, S_{X:foo,main:20} = \{\bot,20,\bot\}, S_{Y:foo,main:20} = \{\bot,20,20\}\}$	$\{Y : foo_{main:20}, Main_0\}$
$Main_0$	$\{\}$	$\{S_{main,0} = \{\}, S_{X:foo,main:16} = \{10,20,10\}, S_{Y:foo,main:16} = \{10,20,20\}, S_{X:foo,main:20} = \{\bot,20,\bot\}, S_{Y:foo,main:20} = \{\bot,20,20\}\}$	$\{\}$

Table 1: Solving the above example using the proposed Interprocedural Constant Propagation.

0.6 Effect of Parallelism on Context Sensitive Constant Propagation Algorithm

In the analysis of flow sensitive context sensitive interprocedural constant propagation algorithm.

0.6.1 Java Program

In the environment such as Java where two threads can run in parallel rather than two tasks (like X10). Our proposed algorithm will fail for one of the particular case because global variables is treated as parameters while maintaining the CP information for a method. Condition required for the algorithm to fail:

- Atleast two method (m_1, m_2) should run in parallel.
- Atleast one of the method (m_1) writes to global variable and other method (m_2) reads the global variable.
- If m_1 updates the global variable consecutively and m_2 misses the atleast one of the write value.

As a example, Let m_1 and m_2 initially both sees the global variable gv with value 10. If both methods are running in parallel and m_1 updates the value of gv twice (first with 20 then with 10). If m_2 misses the $gv = 20$ write, then it's gv will be 10 before and after both write by m_1 and it will never know that gv was updated with value 20. So any variable in m_2 which is dependent on gv will never be analysed with $gv = 20$.

0.6.2 X10 Program

If the environment is X10, then our proposed algorithm will fail.

```
/*
* Program: report.java
* Functionality: X10 program where the proposed algorithm will fail.
* Author: Ravindra kumar
* Last modifier: 25-April-2022
* Bugs: None
 */

int a;
finish{
    async{
        a = 20;
    }
    a = 30;
}
b = a;
```

If we are running the whole program sequentially i.e. ignoring that "async" statement completely, then we will get $b = 30$, using the algorithm proposed but actually it should be $\perp$.

0.7 Modification in the Context Sensitive Constant Propagation

0.7.1 Java program

One of the easy solution is totally ignore the global variable and initialize all global variable with $\perp$.
Another solution is to restrict the write to global variable twice before all method see the intermediate value. This can be done by adding a new list of flags for each global variables. Any write to global variable should be followed with setting the flag for that variable if value differ and both update should be atomic. Before updating the global variable check if it's flag is set if yes, then add all the methods to the current worklist with current value of global variable and set the flag to false before updating the global variable for second time.

0.7.2 X10 program

Problem is we are ignoring the "async" and "finish" statements. All the statements which are running alone should have a definite value and the statements which are running in parallel can have any value. To solve the problem there are two techniques.

- Add control flow edges from all the statements inside the async block to all the statements to whom it will run in parallel with. Add the reverse edge from all the statements to whom it will run in parallel with to all the statements inside the async block. This will ensure the correctness of the program as we have added all the control flow possible. This result in exponential growth of control edges and compute time.

- Add a control edge from the pred of **async statement** to the successor of **async block**. Add another control edge from last statement(s) of **async block** to the first statement after the immediate finish block. This ensures that all the variables after the finish statement will see the correct values. Statements which are running in parallel inside a finish can see any value.

Since second technique is optimized way of analysing the X10 program. We will be using it. Not that statements which are running in parallel will be synchronized at the end of parallelism. All parallel executing statements will not see each other effects.
As we can add the control edges statically. Therefore we can divide the analysis in two phase to avoid the change of previously proposed algorithm.

- Add new control edges.

- Use the proposed algorithm.

Note that we can add the control edges during the analysis too (whenever async statement is encountered, add the corresponding control edges).

0.7.3 Solving X10 example using new technique

```
1  /*
2  * Program: report.java
3  * Functionality: X10 program.
4  * Author: Ravindra kumar
5  * Last modifier: 13-May-2022
6  * Bugs: None
7   */
8
9  int a;
10 finish{
11     async{
12         a = 20;
13     }
14     a = 30;
15 }
16 b = a;
```

Now, In this example, in first phase of the technique, add a control flow edge from line 10 to line 14 and add another edge from line no. 12 to line no. 16. In phase two, this will cause the $m_{16} = m'_{12} \sqcap m'_{14}$ which will give $m_{16}(a) = \perp$ and therefore the $m'16(b) = \perp$. Hence resulting in correct output. Note: m_l, m'_l means m and m' map for line number l.

0.7.4 Some Examples

```
1  /*
2  * Program: report.java
3  * Functionality: X10 program CP analysis. Example_1
4  * Author: Ravindra kumar
5  * Last modifier: 13-May-2022
6  * Bugs: None
7   */
8  class report{
```

```
9       public static void main(String []args){
10          X x = new X();
11
12          int a;
13          int b;
14          int c;
15          int d;
16          finish{
17              b = 40;
18              async{
19                  a = 10;
20                  b = 20;
21              }
22              a = 30;
23              c = x.foo(a,b);
24          }
25          d = x.foo(a,b);
26      }
27  }
28  class X{
29      int foo(int p,int q){
30          return p;
31      }
32  }
```

Currently method Analysed	Current Worklist (Before analysis)	Summary Map (After analysis)	updated Worklist (After analysis)
Initialization	$\{\}$	$\{S_{main,0} = \{\}, S_{X:foo,main:23} = \{\top,\top,\top\}, S_{X:foo,main:25} = \{\top,\top,\top\}$	$\{Main_0\}$
$Main_0$	$\{\}$	$\{S_{main,0} = \{\}, S_{X:foo,main:23} = \{30,40,\top\}, S_{X:foo,main:25} = \{\bot,\bot,\top\}$	$\{X : foo_{main:23}, X : foo_{main:25}\}$
$X : foo_{main:23}$	$\{X : foo_{main:25}\}$	$\{S_{main,0} = \{\}, S_{X:foo,main:23} = \{30,40,30\}, S_{X:foo,main:25} = \{\bot,\bot,\top\}\}$	$\{X : foo_{main:25}, Main_0\}$
$X : foo_{main:25}$	$\{Main_0\}$	$\{S_{main,0} = \{\}, S_{X:foo,main:23} = \{30,40,30\}, S_{X:foo,main:25} = \{\bot,\bot,\bot\}\}$	$\{Main_0\}$
$Main_0$	$\{\}$	$\{S_{main,0} = \{\}, S_{X:foo,main:23} = \{30,40,30\}, S_{X:foo,main:25} = \{\bot,\bot,\bot\}\}$	$\{\}$

Table 2: Solving the above example using the proposed Interprocedural Constant Propagation and async edges.

Therefore at the end c and d variables inside *main* method will have 30 and $\bot$ values correspondingly.

```
/*
* Program: report.java
* Functionality: X10 program CP analysis. Example_2
* Author: Ravindra kumar
* Last modifier: 13-May-2022
* Bugs: None
 */
class report{
    public static void main(String []args){
        X x = new X();
        Y y = new Y();
        int a;
        int b;
        int c;
        int d;
        finish{
            b = 40;
            async{
                a = 10;
                b = 20;
            }
            a = 30;
            c = x.foo(a,b);
        }
        d = y.foo(b,c);
    }
}
class X{
    int foo(int p,int q){
        return p;
    }
}
class Y{
    int foo(int p,int q){
        return q;
    }
}
```

Currently method Analysed	Current Worklist (Before analysis)	Summary Map (After analysis)	updated Worklist (After analysis)
Initialization	{}	$\{S_{main,0} = \{\}, S_{X:foo,main:23} = \{\top,\top,\top\}, S_{Y:foo,main:25} = \{\top,\top,\top\}$	$\{Main_0\}$
$Main_0$	{}	$\{S_{main,0} = \{\}, S_{X:foo,main:23} = \{30,40,\top\}, S_{Y:foo,main:25} = \{\bot,\top,\top\}$	$\{X : foo_{main:23}, Y : foo_{main:25}\}$
$X : foo_{main:23}$	$\{Y : foo_{main:25}\}$	$\{S_{main,0} = \{\}, S_{X:foo,main:23} = \{30,40,30\}, S_{Y:foo,main:25} = \{\bot,\top,\top\}\}$	$\{Y : foo_{main:25}, Main_0\}$
$Y : foo_{main:25}$	$\{Main_0\}$	$\{S_{main,0} = \{\}, S_{X:foo,main:23} = \{30,40,30\}, S_{Y:foo,main:25} = \{\bot,\top,\top\}\}$	$\{Main_0\}$
$Main_0$	{}	$\{S_{main,0} = \{\}, S_{X:foo,main:23} = \{30,40,30\}, S_{X:foo,main:25} = \{\bot,30,\top\}\}$	$\{Y : foo_{main:25}\}$
$Y : foo_{main:25}$	{}	$\{S_{main,0} = \{\}, S_{X:foo,main:23} = \{30,40,30\}, S_{Y:foo,main:25} = \{\bot,30,30\}\}$	$\{Main_0\}$
$Main_0$	{}	$\{S_{main,0} = \{\}, S_{X:foo,main:23} = \{30,40,30\}, S_{X:foo,main:25} = \{\bot,30,30\}\}$	{}

Table 3: Solving the above example using the proposed Interprocedural Constant Propagation and async edges.

Therefore at the end, both c and d variable inside *main* method will have 30 as value.

0.8 Limitation

- Arrays and Pointers are treated as not evaluatable statements.
- No cyclic calling of the procedures.

0.9 Conclusion

The proposed algorithm can be easily shown that it is monotone i.e. each variable is assigned twice and each variable lattice value moves from $\top$ towards the $\bot$ of the lattice. It uses the control flow and call graph information. Analysing the method calls from each call site gives more precise information as compared to context insensitive analysis. The above algorithm can be easily converted to flow insensitive context sensitive analysis by just maintaining a single map for the whole procedure and initialize it with the summary of the current method before analysing the current method. The above interprocedural constant propagation can be made more precise using the Wegmen's et al.'s algorithm i.e. analysing only the path taken/can be taken by branch conditions.

Bibliography

[1] J. Kam and J. Ullman., "Monotone data flow analysis frameworks." *Acta Informatica*, 1977.

[2] G. Kildall, "A unified approach to global program optimization," *Proceedings of Twelfth POPL.*, pp. 194–206, 1973.

[3] M. Wegman and F. K. Zadeck, "Constant propagation with conditional branches," *Proceedings of Twelfth POPL.*, 1985.

[4] K. K. L. T. David Callaham, Keith D. Cooper, "Interprocedural constant propagation," *ACM*, 1986.

[5] R. S. J. U. U. A. Aho, Monica S. Lam, in *Compilers-Princilpe, Techniques, Tools.* Pearson, Boston New York, 2007, pp. 632–638.