CS6843-Program Analysis **22-02-2017**

Constant Propagation

Scribe: Tejas and Anuj

1.1 Introduction

Constant propagation is the process of substituting the values of known constants in expressions at compile time. At every program point, we want to determine the variables that have a constant value. Also, we intend to propagate these calculated or determined values as far forward in program as possible. Constant propagation has long been used in compiler optimization passes in order to turn variable reads and computations into constants, wherever possible.

Definition: A variable $x \in V$ has a constant value $c \in C$ 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.

Consider: $L : x = c;$, where c is a constant
We want to replace the occurrence of x by c, which later might lead to elimination of L altogether. But this kind of optimization is not always correct. We need to keep track of any branch and reassignment which might occur between declaration of a variable and its use.

1.2 Lattice for constant propagation

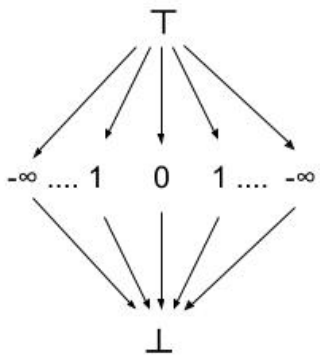

Figure 1.1: Constant Propagation lattice

$\top$: May be constant(undetermined)
$\bot$: Constant value cannot be guaranteed

Given a variable x and a program point u $\top$ to indicate that no definition of x has been seen along any path reaching u. $\bot$ to indicate that x can have different values at u along different paths reaching u
The structure of the lattice is governed by the choice of ignoring those control flow paths along which no definition of the variable has been seen.

Conservative approach.

Worklist structure: Every statement or set of statements of the program corresponds to element in the worklist.

Meet rules :
$\forall x$,
$x \sqcap \top = x$
$x \sqcap \bot = \bot$
$c_1 \sqcap c_1 = c_1$, where c_1 is a constant
$c_2 \sqcap c_1 = \bot$, where c_1 and c_2 are constant

1.3 Flow Function

Consider each statement as basic block. At each program point, we need to maintain a map(M), consisting of variables mapping to a value, $m \in M, v \to m(v)$ where $m(v)$ is a value from lattice ($\top$ or $\bot$ or *constant*).
Let $m, m' \in M$
Flow function F: set of functions. f_s is flow function for statement s : $m' = f_s(m)$
Computing flow function:

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 $f_s(m) = m'$, where:

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

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

- If the RHS is an expression that cannot be evaluated, then, $m'(v) = \bot$

end

At a merge point, get a meet of the flow maps.

1.4 Algorithms

1.4.1 Kildall's Simple Constant algorithm

It uses the program control flow graph for propagation of values. We initialize the map to top. Till the time there is no change in the value of the maps of each block we will perform the operation. We will maintain a *worklist* which will have all the blocks in the program and marked as unvisited.

For all the unvisited successors of block currently in consideration, we will determine the values of maps using flow function and add it to *worklist*.

Simple Constant

```
M[entry] = init
do
    change = false
    worklist ← all statements ∀b,visited(b) = false
    while worklist not empty do
        b = worklist.remove
        visited(b) = true
        m' = f_b(m)
        ∀b' ∈successors of b
        if visited(b') then
            continue
        end
        else
            m[b'] ⊓ = m'
            if m[b'] changes then
                change = true
            end
            worklist.add(b')
        end
    end
while (change == true);
```

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. A reasonable approximation is twice N, since conditional statements typically have only two successors.
V is the number of variables in the program

Complexity:

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

Consider the following example:

```
1  x=10;
2  y=1;
3  z=1;
4  while(x>1){
5        y=x*y;
6        x=x-1;
7        z=z*z;}
8  p=x+y+z;
```

Listing 1: SC example

	Iteration 1 {m(x), m(y), m(z), m(p)}	**Iteration 2** {m(x), m(y), m(z), m(p)}	**Iteration 3** {m(x), m(y), m(z), m(p)}
0. Entry	{⊤, ⊤, ⊤, ⊤}	{⊤, ⊤, ⊤, ⊤}	{⊤, ⊤, ⊤, ⊤}
1. x=10;	{10, ⊤, ⊤, ⊤}	{10, ⊤, ⊤, ⊤}	{10, ⊤, ⊤, ⊤}
2. y=1;	{10, 1, ⊤, ⊤}	{10, 1, ⊤, ⊤}	{10, 1, ⊤, ⊤}
3. z=1;	{10, 1, 1, ⊤}	{10, 1, 1, ⊤}	{10, 1, 1, ⊤}
4. while(x>1){	{10, 1, 1, ⊤}	{⊥, ⊥, 1, ⊤}	{⊥, ⊥, 1, ⊤}
5. y=x*y;	{10, 10, 1, ⊤}	{⊥, ⊥, 1, ⊤}	{⊥, ⊥, 1, ⊤}
6. x=x-1;	{9, 10, 1, ⊤}	{⊥, ⊥, 1, ⊤}	{⊥, ⊥, 1, ⊤}
7. z=z*z; }	{9, 10, 1, ⊤}	{⊥, ⊥, 1, ⊤}	{⊥, ⊥, 1, ⊤}
8. p=x+y+z;	{10, 1, 1, 12}	{⊥, ⊥, 1, ⊥ }	{⊥, ⊥, 1, ⊥ }
change	*true*	*true*	false

Figure 1.2: Simple Constant execution on Listing 1

Explanation- Simple Constant(Figure 1.2): In the given example: The entry block marks the beginning. Block 1 assigns 10 to x as per the flow function. Similarly, y and z will get assigned to 1. The while loop, not being an assignment statement, will have the same map as its predecessor. Now, block 4 has two successor. both will get evaluated and the mapped values get changed accordingly. So, block 8 will assign 12 to p. The value of y,x,z will be updated accordingly for the blocks 5,6,7. This will mark the end of iteration 1.
The second iteration will be same till block 4. Lattice values of x and y will get changed to $\perp$, because predecessors(block 3 and 7) in Iteration 1 have different lattice value and this information will be propagated to its successors(block 5 and 8).
Once the fixed point is reached(in Iteration 3), we need to propagate the constants, wherever possible using the maps. So, the variables on the right hand side of the assignment statement, which turns out to be constant, can be replaced by the constant lattice value they possess at the fixed point. In the given example, the lattice value of x, y, z is $10, 1, 1$, respectively up to statement 3. Any occurrence of these variables is replaced by their respective constant value. From block 4 onwards, the values becomes $\perp, \perp, 1$. So we can replace only z with 1 and not x,y. Hence, for block 7, $z = z * z$, we can replace z by 1. Hence it becomes $z = 1$. Also, block 8 becomes $p = x + y + 1$.

1.4.2 Conditional Constant

Wegbreit's algorithm: With each block maintain a flag called executable. Initially each block is marked not executable. The same simple constant propagation algorithm would be applied here with few minor changes.. We will mark executable as true if only 1 successor is present which implies it is not a conditional instruction. If it has 2 or more instructions which implies it is conditional instruction then we will evaluate the condition and based on the evaluation we will mark the executable flag of its successors. We will update the maps of successor(s) accordingly.

Conditional Constant

```
M[entry] = init
change = false
∀ basic block,b, executable(b) = false
executable(entry) = true
do
    change = false
    worklist ← all statements
    ∀b, visited(b) = false

    while worklist not empty do
        b = worklist.remove
        visited(b) = true
        m' = f_b(m)
        ∀b' ∈successors of b

        if visited(b') then
            continue
        end
        else
            if only one successor(say b1) then
                executable(b1) = true
                m[b1] ⊓ = m'
                if m[b1] changes then
                    change = true
                end
                worklist.add(b1)
            end
            else
                evaluate the condition expression v
                if v == ⊥ then
                    mark all successors as executable
                    m[b'] ⊓ = m'
                    if m[b'] changes then
                        change = true
                    end
                    worklist.add(b')

                end
                if v == constant then
                    mark corresponding successor (b'') as executable
                    m[b''] ⊓ = m'
                    if m[b''] changes then
                        change = true
                    end
                    worklist.add(b'')

                end
            end
        end
    end
while (change == true);
```

Complexity: This algorithm is able to ignore any definition that reaches a use via a program flow graph edge that is never executed. Thus, this algorithm accomplishes a form of dead code elimination called unreachable code elimination. This algorithm has the same asymptotic running time as Simple Constant, $O(E * V^2)$ in the worst case in which no branches are found to be constant. This algorithm is expected to have better average-case complexity, however, since it can ignore parts of the program that will never be executed and since SC should behave better than $O(E * V^2)$

Example 2 - Conditional Constant:

```
1  i=1;
2  if(i==1)
3    j=2;
4  else
5     j=3;
6  printf(i,j);
7  if(j!=2)
8        k=3;
9  else
10       k=4;
11 printf(i,j,k);
```

Listing 2: Conditional Constant Example 2

	Iteration 1 {m(i), m(j), m(k)} , flag	**Iteration 2** {m(i), m(j), m(k)} , flag
0. Entry	{⊤, ⊤, ⊤} executable	{⊤, ⊤, ⊤} executable
1. i=1;	{1, ⊤, ⊤} executable	{1, ⊤, ⊤} executable
2. if(i==1);	{1,⊤, ⊤} executable	{1,⊤, ⊤} executable
3. j=2;	{1, 2, ⊤} executable	{1, 2, ⊤} executable
5. j=3;		
6. printf(i,j);	{1, 2, ⊤} executable	{1, 2, ⊤} executable
7. If(j!=2)		
8. k=3;		
10. k=4;	{1, 2, 4} executable	{1, 2, 4} executable
11. printf(i,j,k);	{1, 2, 4} executable	{1, 2, 4} executable
change	*true*	*false*

Figure 1.3: Conditional Constant Execution: Listing 2

Explanation: Conditional Constant(Figure 1.3)
In conditional constant algorithms, the lattice values will be calculated in similar way as that of Simple Constant, but the blocks dependent on conditional blocks will be considered according to the condition evaluated in parent block. Blocks 5,7,8 will not be set to *executable* in Iteration 1 as condition is not met with.In Iteration 2, as the fixed point is reached, we can propagate the constants now. At block 2, since it will always evaluate to *true*, we can remove the condition and *else* block and replace if-else blocks with $j = 2$. At block 7, since it will always evaluate to *false*, we can remove the if-else blocks(7,8,9,10) and replace with the *else* body(block 10) i.e $k = 4$. Hence the transformed program would be :

```
1  i=1;
2  j=2;
3  printf(i,j);
4  k=4;
5  printf(i,j,k);
```

Listing 3: Listing 2 after Constant Propagation

Example 3 - Conditional Constant:

```
1  i=0;
2  do{
3   if(i==1)
4       j=j+1;
5   else
6       j=2;
7   i=i+1;
8  }while(i<2)
9  printf(i,j);
```

Listing 4: Conditional Constant Example 3

	Iteration 1 {m(i), m(j)} flag	**Iteration 2** {m(i), m(j)} flag	**Iteration 3** {m(i), m(j)} flag	**Iteration 4** {m(i), m(j)} flag
Entry	{⊤, ⊤} executable	{⊤, ⊤} executable	{⊤, ⊤} executable	{⊤, ⊤} executable
1	{0, ⊤} executable	{0, ⊤} executable	{0, ⊤} executable	{0, ⊤} executable
3		{ ⊥, 2} executable	{⊥, ⊥} executable	{⊥, ⊥} executable
4		{ ⊥, 3} executable	{⊥, ⊥} executable	{⊥, ⊥} executable
6	{1, 2} executable	{ ⊥, 2} executable	{⊥, ⊥} executable	{⊥, ⊥} executable
7	{1, 2} executable	{⊥, ⊥} executable	{⊥, ⊥} executable	{⊥, ⊥} executable
8	{1, 2} executable	{⊥, ⊥ } executable	{⊥, ⊥} executable	{⊥, ⊥} executable
9	{1, 2} executable	{⊥, ⊥ } executable	{⊥, ⊥} executable	{⊥, ⊥} executable
change	*true*	*true*	*true*	*false*

Figure 1.4: Conditional Constant Execution: Listing 4

Explanation: Conditional Constant(Figure 1.4)
In conditional constant algorithms, the lattice values will be calculated in similar way as that of simple constant, but the blocks dependent on conditional blocks will be considered according to the condition evaluated in parent block. Blocks 3 and 4 will not be set to executable in Iteration 1 as condition($i == 1$) is false.
In Iteration 2, the lattice value of i is gets changed to $\perp$ at block 3 because lattice values of i in parent blocks(1 and 8) are not same. And this new lattice value gets propagated to all the successors. As a result of i being $\perp$, all the successors of block 3 will become executable.
j will also get assigned to $\perp$ because of meet operation at block 7, where value of j is different for the predecessor blocks(4 and 6).
Once the fixed point is reached, we can propagate the constants based on the maps of each variables at each program point.In the given example, from block 3 onwards, at each program point, the value of maps of both the variables is $\perp$, so, we can not replace these variables. Hence, the transformation will not happen in this case, because the lattice values is $\perp$ and it won't get replaced.

1.5 Applications

- Expressions evaluated at compile time need not be evaluated at run time. If such expressions are inside loops, a single evaluation at compile time can save many evaluations at run time.
- Unreachable code can be detected by identifying conditional branches that always take one of the possible branch paths.
- Since many of the parameters to procedures are constants, using constant propagation with procedure integration can avoid the expansion of code that often results from naive implementations of procedure integration.

References

- Data Flow Analysis: Theory and Practice, Khedker, Sanyal, Karkare
- Constant Propagation with Conditional Branches, Mark N. Wegman and F. Kenneth Zadeck