Putting Data Flow Analysis to Work

Last Time

Iterative Worklist Algorithm via Reaching Definitions

- Why it terminates.
- What it computes.
- Why it works.
- How fast it goes.

Today

- Live Variable Analysis (backward problem)
- Constant Propagation: A Progression in Analysis

Live Variable Analysis

Can a variable v at a point p be used before it is redefined along some path starting at p?

USE(p) - the set of variables that *may* be used before they are defined by this statement or basic block .

DEF(p) - the set of variables that may be defined by this statement or basic block.

```
0:    read I
1:    read N
2:    call check (N)
3:    I = 1
4:    while (I < N) do
5:        A(i) = A(i) + I
6:        I = I + 1
7:    endwhile
8:    print A(N)
```

Live Variable Analysis

A backward data flow problem: *For each point p in the program and each variable x, determine whether x can be used before being redefined along some path starting at p.*

For a basic block, x is *live* if it is used before being redefined within that block, or if it is *live* going out of the block. IN(v) is the set variables live coming into a block, and OUT(v) is the set of variables live going out of a block.

> USE(v): x $\in$ USE(v) iff x may be used before it is defined in v
>
> DEF(v): x $\in$ DEF(v) iff x must be defined before it is used in v (DEF(v) == KILL(v))
>
> $\text{OUT}(v) = \bigcup_{s \in \text{SUCC}(v)} \text{IN}(s)$
>
> $\text{IN}(v) = \text{USE}(v) \bigcup (\text{OUT}(v) - \text{DEF}(v))$

The **monotone data flow framework** uses powerset of X (all variables) lattice. The transfer function

$$T_v(x) = \text{USE}(v) \bigcup (x - \text{DEF}(v))$$

The meet is set union.

The operation space is monotone and distributive, therefore the solution will result in the MOP solution.

Work List Iterative Algorithm Rehashed

initialization
worklist $\leftarrow$ the set of all nodes
while(*worklist* $\neq \emptyset$)
    pick and remove a node n from *worklist*
    recompute Data Flow Equations
    if the answer changed then
        add affected nodes to *worklist*

Initialization:

> OUT(v)
>
> IN(v)

Data flow equations:

> OUT(v)
>
> IN(v)

Live Variable Algorithm

Algorithm:

for all v
    $\text{OUT}(v) = \emptyset$
    $\text{IN}(v) = \text{USE}(v)$
endfor
worklist $\leftarrow$ the set of all nodes
while (*worklist* $\neq \emptyset$)
    pick and remove a node v from *worklist*
    $\text{OUT}(v) = \bigcup_{s \in \text{SUCC}(v)} \text{IN}(s)$
    oldin $= \text{IN}(v)$
    $\text{IN}(v) = \text{USE}(v) \cup (\text{OUT}(v) - \text{DEF}(v))$
    if *oldin* $\neq \text{IN}(v)$
        worklist $\leftarrow$ *worklist* $\cup$ $\text{PRED}(v)$
end while

Live Variable Example

Can a variable v at a point p be used before it is redefined along some path starting at p?

USE(p) - the set of variables that *may* be used before they are defined by this statement or basic block.

DEF(p) - the set of variables that may be defined by this statement or basic block.

		USE	DEF	LIVE
0:	read I			
1:	read N			
2:	call check (N)			
3:	I = 1			
4:	while (I < N) do			
5:	A(i) = A(i) + I			
6:	I = I + 1			
7:	endwhile			
8:	print A(N)			

Constant Propagation

Discover variables and expressions that are constant and propagate them as far forward through the program as possible.

Uses:

- Evaluates expressions at compile time instead of runtime.
- Eliminate *dead code*, code that can never be executed, *e.g.*, debugging code.
- Improves the effectiveness of many optimizations, *e.g.*, value numbering, software pipelining.

Since it is an analysis, there are no disadvantages.

Constant Propagation

Lattice:

$$\top$$

$$\ldots -3 \quad -2 \quad -1 \quad 0 \quad 1 \quad 2 \quad 3 \quad \ldots$$

$$\perp$$

Meet Rules:

$$\begin{array}{rclll} a & \cap & \top & \rightarrow & a & \\ a & \cap & \perp & \rightarrow & \perp & \\ \textit{constant} & \cap & \textit{constant} & \rightarrow & \textit{constant} & \text{(if equal)} \\ \textit{constant} & \cap & \textit{constant} & \rightarrow & \perp & \text{(if not equal)} \end{array}$$

Optimistic assumption: all variables start at an unknown constant value ($\top \neq \perp$)

Pessimistic assumption: all variables are not constant ($\top = \perp$)

Kildall's algorithm for Constant Propagation

Using a worklist

1. Add successor basic blocks (statements) of *start*.
2. Given $v =$ *expression*,
3. For every use w in *expression*, find the reaching definitions. If all constant, set value of w to the constant value, otherwise set value of $w = \perp$.
4. If any w is $\perp$, the set value of $v = \perp$.
   $\Longrightarrow$ *What is the effect of a pessimistic vs. optimistic assumption?*
5. Otherwise, if they are all constants, set v to the value of the expression.

Simple constants. *No information is assumed about which direction branches take, and only one value for each variable is maintained along each path in the program.*

Time: $O(E * V^2)$ - $E * V$ node visits, V operations at a visit.

Space: $O(N * V)$, N statements in the program

Kildall, G. A., A unified approach to global program optimization. *conference Record of the First ACM Symposium on Principles of Programming Languages,* October 1973, pages 194-206.

DefUse Graph

Reaching definitions and live variables help us find constants.

A *DefUse* chain is a connection from a *definition site* for a variable to a *use site* for that variable along a path in the *CFG* without passing through another definition.

How should we build it?

$z =$ $\qquad$ $z =$

$x =$ $\qquad$ $x =$

$y =$ $\qquad y = z$ $\qquad y =$ $\qquad y = z$

$x =$ $\qquad$ $x =$

$= x + y + z$ $\qquad$ $= x + y + z$

DefUse edges $\qquad$ *DefJoin* and *JoinUse* edges

Hints of static single assignment (SSA)

Reif and Lewis Constant Propagation

Finds **simple constants**, but improves the time and space complexity of Kildall.

Worklist algorithm:

1. Put all the root edges from the *DefUse* graph on the worklist.
2. A definition site in the roots, is assigned a constant, if it can be evaluated to a constant, otherwise it is assigned $\perp$.
3. All other variables are assigned $\top$.
4. *DefJoin* edges are taken off the worklist. The value of the src of an edge is propagated to the use using the meet rules.
5. If the value is lowered, the new value is propagated and evaluated at the use expression. If this causes a variable to be lowered, the node is added to the worklist.

Time: The complexity is now in terms of the DefUse graph.

Reif and Lewis Constant Propagation

$i = 1$
for ()
    $j = i$
    $i = f(\ldots)$
    $i = j$
endfor

Handles loops because of optimistic assumption. Previous techniques couldn't do loops.

What do we need to do to detect that j is constant?

$i = 1$
$j = 3$
$\ldots$
if $(i == 1)$
    then $j = 1$
$k = j$

Wegman and Zadeck - Conditional Definition

Conditional Definition. Keeps track of the results of conditional branches. A form of dead code elimination.

- Whenever the expression in a branch is a constant, determine the direction of the branch.
- Only propagate definitions when the flow graph node is marked as executable.
- Use symbolic execution of the program to mark edges.
- When propagating constants ignore edges at join nodes that are not executable.

Wegman, M. N. and Zadeck, F. K., Constant Propagation with Conditional Branches, *Conference Record of the Twelfth Annual ACM Symposium on the Principles of Programming Languages*, January, 1985.

Wegman Zadeck - Conditional Constant

Conditional Constant. Adds identity $i = i$ on *birth points* that are not definitions to determine kills along paths that must be executed.

Birth point for a variable v:

- Each definition site for v is a birth point.
- Let n be a node with two or more incoming edges. If there is a node m which is a birth point for v and there is a birth point free path from m to n along one in edge, but not the other, then n is a birth point for v.

Time: $O(2*N+2*C)$ C is the number of DefUse chains.

Constant Propagation truth tables

Lattice for integer addition, multiplication, mod, etc.

op	$\top$	c_1	$\bot$
$\top$	$\top$	c_1	$\bot$
c_2	c_2	c_1 *op* c_2	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$

Lattice for AND

AND	$\top$	*false*	*true*	$\bot$
$\top$	$\top$	*false*	$\top$	$\bot$
false	*false*	*false*	*false*	*false*
true	$\top$	*false*	*true*	$\bot$
$\bot$	$\bot$	*false*	$\bot$	$\bot$

Lattice for OR

OR	$\top$	*false*	*true*	$\bot$
$\top$	$\top$	$\top$	*true*	$\bot$
false	$\top$	*false*	*true*	$\bot$
true	*true*	*true*	*true*	*true*
$\bot$	$\bot$	$\bot$	*true*	$\bot$

A last refinement

```
if (i = 4) then
    j = i + 2
    ... use of j, i
endif
... use of j, i
```

Insert an assignment/assertion $i = 4$ on the *true* branch.

Worst Case Example

```
select j
    when a do
        i = 1
    when b do
        i = 2
    when c do
        i = 3
end select
select k
    when a do
        a = i
    when b do
        b = i
    when c do
        c = i
end select
```

$i = 1$ $i = 2$ $i = 3$

$a = i$ $b = i$ $c = i$

$i = 1$ $i = 2$ $i = 3$

$a = i$ $b = i$ $c = i$

Next Time

More Program Representation

- Dominators
- Control Dependence

K. D. Cooper, T. Harvey, and K. Kennedy, A Simple, Fast Dominance Algorithm, Software Practice and Experience, 2001:4:1-28.