Constant Propagation

Goal

– Discover constant variables and expressions and propagate them forward through the program

Uses

– Evaluate expressions at compile time instead of run time
– Eliminate dead code (e.g., debugging code)
– Improve efficacy of other optimizations (e.g., value numbering and software pipelining)
Roadmap

1. Simple Constants
   Kildall [1973]
   faster

2. Sparse Simple Constants
   Reif and Lewis [1977]

3. Conditional Constants
   Wegbreit [1975]
   faster

4. Sparse Conditional Constants
   Wegman & Zadeck [1991]
   More constants
Kinds of Constants

**Simple constants** Kildall [1973]
- Constant for all paths through a program

**Conditional constants** Wegbreit [1975]
- Constant for actual paths through a program (when only one direction of a conditional is taken)

```
c := 1
... 
if c=1
```

- true
- false

```
j := 3
```

```
j := 5
```

```
j?
```
Implementing Simple Constant Propagation

**Standard worklist algorithm**
- Identifies simple constants
- For each program point, maintains one constant value for each variable
- $O(EV)$ (E is the number of edges in the CFG; V is number of variables)

**Problem**
- Inefficient, since constants may have to be propagated through irrelevant nodes

**Solution**
- Exploit a sparse dependence representation (e.g., SSA)
**Sparse Simple Constant Propagation**

**Reif and Lewis algorithm** Reif and Lewis [1977]
- Identifies simple constants
- Faster than Simple Constants algorithm

**SSA edges**
- Explicitly connect defs with uses
- How would you do this?

**Main Idea**
- Iterate over SSA edges instead of over all CFG edges

```
x = 1
```

```
y = x
```
Sparse Simple Constants Algorithm (Ch. 19 in Appel)

worklist = all statements in SSA

while worklist ≠ ∅
    Remove some statement S from worklist
    if S is x = phi(c,c,...,c) for some constant c
        replace S with v = c
    if S is x=c for some constant c
        delete s from program
        for each statement T that uses v
            substitute c for x in T
    worklist = worklist union {T}
Sparse Simple Constants

Complexity
- $O(E') = O(EV)$, $E'$ is number of SSA edges
- $O(n)$ in practice
Other Uses of SSA

Dead code elimination

while ∃ a variable v with no uses and whose def has no other side effects

Delete the statement s that defines v

for each of s’s ud-chains

Delete the corresponding du-chain that points to s

\[ x = a + b \]

\[ y = x + 3 \]

If y becomes dead and there are no other uses of x, then the assignment to x becomes dead, too

– Contrast this approach with one that uses liveness analysis
  – This algorithm updates information incrementally
  – With liveness, we need to invoke liveness and dead code elimination iteratively until we reach a fixed point
Other Uses of SSA (cont)

Induction variable identification
- Induction variables
  - Variables whose values form an arithmetic progression
  - Useful for strength reduction and loop transformations

Why bother?
- Automatic parallelization, . . .

Simple approach
- Search for statements of the form, \( i = i + c \)
- Examine ud-chains to make sure there are no other defs of \( i \) in the loop
- Does not catch all induction variables. Examples?
Types of Induction Variables

- **Basic** induction variables
  - Variables that are defined once in a loop by a statement of the form, \( i = i + c \) (or \( i = i \times c \)), where \( c \) is a constant integer

- **Derived** induction variables
  - Variables that are defined once in a loop as a linear function of another induction variable
    - \( j = c_1 \times i + c_2 \)
    - \( j = i / c_1 + c_2 \), where \( c_1 \) and \( c_2 \) are loop invariant
Induction Variable Identification (cont)

Informal SSA-based Algorithm

– Build the SSA representation
– Iterate from innermost CFG loop to outermost loop
  – Find SSA cycles
    – Each cycle may be a basic induction variable if a variable in a cycle is a function of loop invariants and its value on the current iteration
  – Find derived induction variables as functions of loop invariants, its value on the current iteration, and basic induction variables
**Induction Variable Identification (cont)**

**Informal SSA-based Algorithm** (cont)

- Determining whether a variable is a function of loop invariants and its value on the current iteration
  - The $\phi$ -function in the cycle will have as one of its inputs a def from inside the loop and a def from outside the loop
  - The def inside the loop will be part of the cycle and will get one operand from the $\phi$ -function and all others will be loop invariant
  - The operation will be plus, minus, or unary minus