A generic worklist analysis algorithm

Maintain a mapping from each program point to info at that point
  • optimistically initialize all pp's to T
Set other pp's (e.g. entry/exit point) to other values, if desired

Maintain a worklist of nodes whose flow functions needs to be evaluated
  • initialize with all nodes in graph

While worklist nonempty do
  Pop node off worklist
  Evaluate node's flow function, given current info on predecessor/successor pp's, allowing it to change info on predecessor/successor pp's
  If any pp's changed, then put adjacent nodes on worklist (if not already there)

For faster analysis, want to follow topological order
  • number nodes in topological order
  • pop nodes off worklist in increasing topological order

It Just Works!

Advanced program representations

Goal:
  • more effective analysis
  • faster analysis
  • easier transformations

Approach:
  • more directly capture important program properties
  • e.g. data flow, independence

Examples

CFG:
  + simple to build
  + complete
  + no derived info to keep up to date during transformations
  − computing info is slow and/or ineffective
    • lots of propagation of big sets/maps

Def/use chains

Def/use chains directly linking defs to uses & vice versa
  + directly captures data flow for analysis
    • e.g. constant propagation, live variables easy
  − ignores control flow
    • misses some optimization opportunities, since it assumes all paths taken
    • not executable by itself, since it doesn’t include control dependence links
    • not appropriate for some optimizations, such as CSE and code motion
  − must update after transformations
    • but just thin out chains
  − space-consuming, in worst case: $O(E^2V)$
  − can have multiple defs of same variable in program, multiple defs can reach a use
    • complicates analysis
Static Single Assignment (SSA) form

[Alpern, Rosen, Wegman, & Zadeck, two POPL 88 papers]

Invariant: at most one definition reaches each use

Constructing equivalent SSA form of program:
1. Create new target names for all definitions
2. Insert pseudo-assignments at merge points reached by multiple definitions of same source variable:
   \[ x_n := \phi(x_1, \ldots, x_n) \]
3. Adjust uses to refer to appropriate new names

Comparison

+ lower worst-case space cost than def/use chains: \( O(EV) \)
+ algorithms simplified by exploiting single assignment property:
  • variable has a unique meaning independent of program point
  • can treat variable & value synonymously
+ transformations not limited by reuse of variable names
  • can reorder assignments to same source variable, without affecting dependences of SSA version

− still not executable by itself
− still must update/reconstruct after transformations
− inverse property (static single use) not provided
  • dependence flow graphs [Pingali et al. ] and value dependence graphs [Weise et al. ] fix this, with single-entry, single-exit (SESE) region analysis

Very popular in research compilers, analysis descriptions
**Common subexpression elimination**

At each program point, compute set of available expressions:
map from expression to variable holding that expression
- e.g. \{a+b \rightarrow x, \ -c \rightarrow y, \ \ast p \rightarrow z\}

CSE transformation using AE analysis results:
if \(a+b \rightarrow x\) available before \(y := a+b\), transform to \(y := x\)

**Specification**

All possible available expressions:
AvailableExprs = \{expr \rightarrow var\} \forall expr \in Expr, \forall var \in Var
- Var = set of all variables in procedure
- Expr = set of all right-hand-side expressions in procedure
[is this a function from Exprs to Vars, or just a relation?]

Domain AV = \(\langle \text{Pow(AvailableExprs)}, \leq_{AV} \rangle\)
\[
\text{ae}_1 \leq_{AV} \text{ae}_2 \iff
\]
- top:
- bottom:
- meet:
- lattice height:

**Constraints**

\[
\text{AE}_x := y \ op \ z:
\]

\[
\text{AE}_x := y:
\]

Initial conditions at program points?
What direction to do analysis?
Can use bit vectors?

**Example**

```
i := a + b
x := i * 4
j := i
i := c
z := j * 4
m := b + a
w := 4 * m
y := i * 4
i := i + 1
```

\[
\text{AE}_x := y:
\]

\[
\text{AE}_x := y:
\]
Exploiting SSA form

Problem: previous available expressions overly sensitive to name choices, operand orderings, renamings, assignments, ...

A solution:

Step 1: convert to SSA form
• distinct values have distinct names
  ⇒ can simplify flow functions to ignore assignments

\[
\text{AE}_{\text{SSA}}: \ x := y \ op \ z
\]

Step 2: do copy propagation
• same values (usually) have same names
  ⇒ avoid missed opportunities

Step 3: adopt canonical ordering for commutative operators
  ⇒ avoid missed opportunities

Example

\[
\begin{align*}
  i &:= a + b \\
x &:= i \times 4 \\
j &:= i \\
i &:= c \\
z &:= j \times 4 \\
y &:= i \times 4 \\
i &:= i + 1 \\
m &:= b + a \\
w &:= 4 \times m \\
i &:= a + b \\
x &:= i \times 4
\end{align*}
\]

After SSA conversion, copy propagation, & operand order canonicalization:

\[
\begin{align*}
  i_1 &:= a_1 + b_1 \\
x_1 &:= i_1 \times 4 \\
  j_1 &:= i_1 \\
i_2 &:= c_1 \\
z_1 &:= i_1 \times 4 \\
i_4 &:= \phi(i_1, i_3) \\
y_1 &:= i_4 \times 4 \\
i_3 &:= i_4 + 1 \\
m_1 &:= a_1 + b_2 \\
w_1 &:= m_1 \times 4
\end{align*}
\]

Loop-invariant code motion

Two steps: analysis & transformation

Step 1: find invariant computations in loop
• invariant: computes same result each time evaluated

Step 2: move them outside loop
• to top: code hoisting
  • if used within loop
• to bottom: code sinking
  • if only used after loop
Detecting loop-invariant expressions

An expression is invariant w.r.t. a loop $L$ iff:

**Base cases:**
- It's a constant
- It's a variable use, all of whose defs are outside $L$

**Inductive cases:**
- It's an idempotent computation all of whose args are loop-invariant
- It's a variable use with only one reaching def, and the rhs of that def is loop-invariant

Computing loop-invariant expressions

**Option 1:**
- Repeat iterative DDA until no more invariant expressions found
- To start, optimistically assume all expressions loop-invariant

**Option 2:**
- Build def/use chains, follow chains to identify & propagate invariant expressions

**Option 3:**
- Convert to SSA form, then similar to def/use form
Loop-invariant expression detection for SSA form

SSA form simplifies detection of loop invariants, since each use has only one reaching definition.

An expression is invariant w.r.t. a loop $L$ iff:

**Base cases:**
- It’s a constant
- It’s a variable use whose single def is outside $L$

**Inductive cases:**
- It’s an idempotent computation all of whose args are loop-invariant
- It’s a variable use whose single def’s rhs is loop-invariant

$\phi$ functions are not idempotent

Example using SSA form

Example using SSA form & preheader

Code motion

When find invariant computation $S: z := x \text{ op } y$, want to move it out of loop (to loop preheader)

**When is this legal?**

**Sufficient conditions:**
- $S$ dominates all loop exits
  
  
  [A dominates $B$ when all paths to $B$ must first pass through $A$]
- otherwise may execute $S$ when never executed otherwise
- can relax this condition, if $S$ has no side-effects or traps, at cost of possibly slowing down program
- $S$ is only assignment to $z$ in loop, & no use of $z$ in loop is reached by any def other than $S$
- otherwise may reorder defs/uses and change outcome
- unnecessary in SSA form!

If met, then can move $S$ to loop preheader
- but preserve relative order of invariant computations, to preserve data flow among moved statements
Example of need for domination requirement

\[
x := 0 \\
y := 1 \\
z := 0? \\
x := a \times b \\
y := x / z \\
q := x + y
\]

Avoiding domination restriction

Requirement that invariant computation dominates exit is strict
- nothing in conditional branch can be moved
- nothing after loop exit test can be moved

Can be circumvented through other transformations such as loop normalization
- move loop exit test to bottom of loop

Example in SSA form

Restrictions unnecessary if in SSA form
- if reorder defs/uses, generate code along merging arcs to implement \( \phi \) functions

Example of data dependence restrictions

“\( S \) is only assignment to \( z \) in loop, & no use of \( z \) in loop is reached by any def other than \( S \)”

Example in SSA form

Restrictions unnecessary if in SSA form
- if reorder defs/uses, generate code along merging arcs to implement \( \phi \) functions
Loop-invariant code copying

Alternative to code motion:
- **copy** instruction to loop header, assigning to new temp, then do CSE & copy propagation to simplify in-loop version
  - more modular design, leverage off of existing optimizations

Can always copy, unless instruction has side-effects
CSE & copy propagation will eliminate in-loop instruction exactly when (non-SSA) loop-invariant code motion would have, PLUS can replace invariant but unmovable instructions with copies

SSA-based code motion gets same effect
- copies correspond to reified $\phi$ functions

Example

Control dependence

Must ensure side-effects occur in proper order
Must ensure side-effects occur only under right conditions

CFG represents control dependence explicitly
- but overspecifies control dependence requirements

Program dependence graph (PDG): data dependence graph + control dependence graph (CDG)
[Ferrante, Ottenstein, & Warren, TOPLAS 87]

Idea: represent controlling conditions directly
- complements data dependence representation

A node (basic block) $N_f$ is **control-dependent** on another $N_2$ iff
$N_2$ determines whether $N_f$ executes, i.e.
- there exists a path from $N_f$ to $N_2$ s.t. every node in the path other than $N_f$ is **post-dominated** by $N_2$
- $N_2$ does not post-dominate $N_f$

Control dependence graph:
- $N_f$ proper descendant of $N_2$ iff $N_f$ control-dependent on $N_2$
  - label each child edge with required branch condition
  - group all children with same condition under **region** node

Two sibling nodes execute under same control conditions ⇒ can be reordered or parallelized, as data dependences allow

Challenging to “sequentialize” back into CFG form
**Example**

1. `y := p + q`
2. `x > NULL?`
3. `a := x * y`
4. `a := y - 2`
5. `w := y / q`
6. `x > NULL?`
7. `b := 1 << w`
8. `r := a % b`

**An example with a loop**

**Value dependence graphs**

[Weise, Crew, Ernst, & Steensgaard, POPL 94]

Idea: represent all dependences, including control dependences, as data dependences

- simple, direct dataflow-based representation of all “interesting” relationships
- analyses become easier to describe & reason about
- harder to sequentialize into CFG

Control dependences as data dependences:

- control dependence on order of side-effects
  - data dependence on reading & writing to global Store
- optimizations to break up accesses to single Store into separate independent chunks (e.g. a single variable, a single data structure)
- control dependence on outcome of branch
  - a select node, taking test, then, and else inputs

Loops implemented as tail-recursive calls to local procedures

Apply CSE, folding, etc. as nodes are built/updated

Like DAG representation of BB, but for whole procedure

**VDG for example, after store splitting**

```
y := p + q
if x > NULL then a := x * y else a := y - 2
w := y / q
if x > NULL then b := 1 << w
r := a % b
```