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: 
   \[ x_i := \phi(x_1, \ldots, x_n) \]
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. \( \{a+b \rightarrow x, -c \rightarrow y, *p \rightarrow z\} \)

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:
- \( \text{AvailableExprs} = \{\text{expr} \rightarrow \text{var} \mid \forall \text{expr} \in \text{Expr}, \forall \text{var} \in \text{Var}\} \)
- \( \text{Var} = \text{set of all variables in procedure} \)
- \( \text{Expr} = \text{set of all right-hand-side expressions in procedure} \)

Domain \( \text{AV} = \langle \text{Pow(AvailableExprs)}, \leq_{\text{AV}} \rangle \)

\( \text{ae}_1 \leq_{\text{AV}} \text{ae}_2 \iff \)
- top:
- bottom:
- meet:
- lattice height:

Constraints

\( \text{AE}_{\text{a}} := y \text{ op } z \)

\( \text{AE}_{\text{x}} := y \)

Initial conditions at program points?

What direction to do analysis?

Can use bit vectors?

Example

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

AE\textsuperscript{SSA}:

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

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

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
Example

\[
\begin{align*}
  x & := 3 \\
  y & := 4 \\
  y & := 5 \\
  z & := x \times y \\
  q & := y \times y \\
  w & := y + 2 \\
  p & := w + y \\
  x & := x + 1 \\
  q & := q + 1 \\
  w & := w + 5 
\end{align*}
\]

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 DFA
  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

Example using def/use chains

\[
\begin{align*}
  x & := 3 \\
  y & := 4 \\
  y & := 5 \\
  z & := x \times y \\
  q & := y \times y \\
  w & := y + 2 \\
  p & := w + y \\
  x & := x + 1 \\
  q & := q + 1 \\
  w & := w + 5 
\end{align*}
\]
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

```
x_1 := 3
y_1 := 4
y_2 := 5
x_2 := \phi(x_1, x_3)
y_3 := \phi(y_1, y_2)
z_1 := x_2 * y_3
q_1 := y_3 * y_3
w_1 := y_3 + 2
w_2 := w_1 + 5
w_3 := \phi(w_1, w_2)
p_1 := w_3 + y_3
x_3 := x_2 + 1
q_2 := q_1 + 1
```

Example using SSA form & preheader

```
x_1 := 3
y_1 := 4
y_2 := 5
y_3 := \phi(y_1, y_2)
x_2 := \phi(x_1, x_3)
y_3 := \phi(y_1, y_2)
z_1 := x_2 * y_3
q_1 := y_3 * y_3
w_1 := y_3 + 2
w_2 := w_1 + 5
w_3 := \phi(w_1, w_2)
p_1 := w_3 + y_3
x_3 := x_2 + 1
q_2 := q_1 + 1
```

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 := a * 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
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

```
x := a * b
y := q * x
q := z * w
q := 0
y := 1
```

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

Control dependence graph

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

Control dependence graph:
- \( N_i \) proper descendant of \( N_j \) iff \( N_i \) control-dependent on \( N_j \)
  - 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 \( \Rightarrow \) can be reordered or parallelized, as data dependences allow

Challenging to "sequentialize" back into CFG form
Example

\begin{align*}
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
\end{align*}

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

An example with a loop

VDG for example, after store splitting

\begin{align*}
y &:= p + q \\
\text{if } x > NULL \text{ then } a &:= x * y \text{ else } a := y - 2 \\
w &:= y / q \\
\text{if } x > NULL \text{ then } b &:= 1 << w \\
r &:= a \% b
\end{align*}