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
  • not too hard (just remove edges)

− space-consuming, in worst case: \(O(E^2 V)\)

− can have multiple defs of same variable in program,
  multiple defs can reach a use
  • complicates analysis

Example
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

Example

\[
\begin{align*}
x := & \ldots \\
y := & \ldots \\
\ldots x \ldots \\
\end{align*}
\]

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 & its contents synonymously
  • can have single global table mapping var to info, not one per program pt.
+ 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

Common subexpression elimination

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

(More generally, can have map from expensive expression to equivalent but cheaper expression
• subsumes CSE, constant prop, copy prop.)

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

Very popular in research compilers, analysis descriptions
Specification

All possible available expressions:

\[
\text{AvailableExprs} = \{ \text{expr} \to \text{var} \mid \forall \text{expr} \in \text{Exprs}, \forall \text{var} \in \text{Vars} \} = \text{Exprs} \times \text{Vars}
\]

- \( \text{Exprs} \) = set of all right-hand-side expressions in procedure
- \( \text{Vars} \) = set of all variables in procedure

[is this a function from \( \text{Exprs} \) to \( \text{Vars} \), or just a relation?]

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

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

Initial conditions at program points?

What direction to do analysis?

Can use bit vectors?
Can summarize sequences of flow functions?

Example

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
  \( \Rightarrow \) can simplify flow functions to ignore assignments

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

Step 2: do \textbf{copy propagation}
- same values (usually) have same names
  \( \Rightarrow \) avoid missed opportunities

Step 3: adopt canonical ordering for commutative operators
  \( \Rightarrow \) avoid missed opportunities
Example

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

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

\[
\begin{align*}
&i_1 := a_1 + b_1 \\
&x_1 := i_1 \times 4 \\
&j_1 := i_1 \\
&i_2 := c_1 \\
&z_1 := i_1 \times 4 \\
&m_1 := a_1 + b_1 \\
&w_1 := m_1 \times 4
\end{align*}
\]