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 only ever remove edges, not add

- space-consuming, in worst case: $O(N^2)$ edges per variable

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:
   \[ x_m := \phi(x_1, \ldots, x_i) \]
3. Adjust uses to refer to appropriate new names

Example

```
x := x + y
y := y + 1
```

Comparison

+ lower worst-case space cost than def/use chains: \(O(EV)\)
+ algorithms simplified by single assignment property:
  - variable has a unique meaning independent of program point
  - can treat variable, its defining stmt, & its value synonymously
  - can have single global table mapping var to info, not one per program pt. that must be propagated, copied, etc.
+ transformations not limited by reuse of variable names
  - can reorder assignments to same source variable, without changing meaning in SSA version

- still not executable by itself
  - and \(\phi\)-functions require an oracle!
  - 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

Implementing \(\phi\)-functions

Semantics of \(x_m := \phi(x_1, \ldots, x_i)\):
set \(x_m\) to \(x_i\), if control last came from predecessor \(i\)

How to implement (generate code for) this?
+ along each predecessor edge \(i\), insert \(x_m := x_i\)
+ delete \(\phi\) statement

If register allocator assigns \(x_m, x_1, \ldots, x_i\) to the same register, then these move instructions will be deleted
+ \(x_m, x_1, \ldots, x_i\) usually have non-overlapping lifetimes, so this kind of register assignment is legal
**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 \(y := x\)

**Specification**

All possible available expressions \(AvailableExpr = Expr \times Var\)

- \(Expr\) = set of all right-hand-side expressions in procedure (or maybe all possible expressions)
- \(Var\) = set of all variables in procedure

[Is this a function from \(Expr\) to \(Var\), or just a relation?]

Domain \(AV = Pow(AvailableExpr)\)

\[ae_1 \leq_{AV} ae_2 \iff\]

\[T_{AV} =\]

\[\perp_{AV} =\]

\[ae_1 \cap_{AV} ae_2 \iff\]

\[\text{height}(AV) =\]

**Flow functions**

What direction to do analysis?

Initial conditions?

\[AE_X := y \in \mathbb{C} \ni (m) =\]

\[AE_Y := y \in \mathbb{C} \ni (m) =\]

**Example**

```
    i := a + b
    x := i * 4

    j := i
    i := c
    z := j * 4

    m := b + a
    x := 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

\[ AX := Y \forall \nexists (m) \]

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:

 SSA form and pointers

What about pointers?
\[ x := 5; \]
\[ y := 7; \]
\[ p := \text{new int}; \]
\[ q := \text{test}1 ? \& x : (\text{test}2 ? \& y : p); \]
\[ *q := 9; \]
// what are the unique SSA names for \( x \) & \( y \) here? \( *p? \)
\[ x := x + 1; \]
// what does \( q \) point to here?

SSA wishes to assign a unique name for each variable (memory location?) at each point
- dynamic memory allocations introduce many “anonymous variables”
- pointer stores don’t definitely update any variable, but may update many
- SSA gives different names to the same variable, but \& creates a pointer to all of them
Some solutions

Option 1: don’t use SSA invariant for pointed-to memory
- heap memory, variables that have their addresses taken

Option 2: insert copies between SSA vars and real vars
before and/or after may-use/may-def operations
- pointers point to real, non-SSA variable
- insert \texttt{var} := \texttt{var} \texttt{before any may-use/may-def of} \texttt{var}
- insert \texttt{var} := \texttt{test} \texttt{after any may-def of} \texttt{var}
  \texttt{(test \texttt{var}) uses oracle to return either} \texttt{var} \texttt{or} \texttt{var}

\begin{verbatim}
x_1 := 5;  
y_1 := 7;
p_1 := \text{new int};  
q_1 := \text{test} \texttt{? &x : test} \texttt{? &y : p_1});  
  x := x_2;  
y := y_2;  
*q_1 := 9;  
x_2 := (x_2, x);  
y_2 := (y_2, y);  
x_3 := x_2 + 1;
\end{verbatim}

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

```
x := 3
y := 4
y := 5
```

```
z := x * y
q := y * y
w := y + 2
```

```
w := w + 5
```

```
p := w + y
x := x + 1
q := q + 1
```

Detecting loop-invariant expressions

An expression is invariant w.r.t. a loop \texttt{L} iff:

(base cases:)
- it’s a constant
- it’s a variable use, all of whose def\textminuses are outside \texttt{L}

(inductive cases:)
- it’s a pure 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

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 a pure 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 pure

Example using SSA form
Example using SSA form & preheader

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

Code motion

When find invariant computation \( S: z := x \text{ op } y \),
want to move it out of loop (to loop preheader)
- preserve relative order of invariant computations,
to preserve data flow among moved statements

When is this legal?

Condition #1: domination restriction

To move \( S: z := x \text{ op } y \),
\( S \) must dominate all loop exits
\( [A \text{ dominates } B \text{ when all paths to } B \text{ first pass through } A] \)
- otherwise may execute \( S \) when never executed otherwise
- if \( S \) is pure, then can relax this condition,
at cost of possibly slowing down program

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

Avoiding domination restriction

Requirement that invariant computation dominates exit is strict
- nothing inside a conditional branch can be moved
- nothing after a loop exit test can be moved
- what happens in a while loop? a for loop?

Can be circumvented through other transformations
such as loop normalization
- move loop exit test to bottom (while-do \( \Rightarrow \) if-do-while)

Before
\[ i := 0 \]
\[ i < N? \]
\[ x := a \div b \]
\[ i := i + 1 \]

After
\[ i := 0 \]
\[ i < N? \]
\[ x := a \div b \]
\[ i := i + 1 \]
\[ i < N? \]
**Condition #2: data dependence restriction**

To move $S$: $z := x \text{ op } y$,
- $S$ must be the only assignment to $z$ in loop, and
- no use of $z$ in loop is reached by any def other than $S$
  - otherwise may reorder defs/uses and change outcome

- $z := 0$
- $z := z + 1$
- $z := 5$

- \[ S \]

**Avoiding data dependence restriction**

Restriction unnecessary if in SSA form
- implementation of $\phi$ functions as moves will cope with reordered defs/uses

- $z_1 := 5$
- $z_2 := \phi(z_1, z_4)$
- $z_3 := z_2 + 1$
- $z_4 := 0$

- \[ S \]

**More refined representations**

Problem: control-flow edges in CFG overspecify evaluation order

Solution: introduce more refined notions w/ fewer constraining edges that still capture required orderings
- side-effects occur in proper order
- side-effects occur only under right conditions

Some ideas:
- explicit control dependence edges,
  - control-equivalent regions,
  - control-dependence graph (PDG)
- operators as nodes (Click, VDG, Whirlwind, etc.)
  - computable $\phi$-function operator nodes
- control dependence via data dependence (VDG)

**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) $Y$ is **control-dependent** on another $X$ iff $X$ determines whether $Y$ executes, i.e.
- there exists a path from $X$ to $Y$ s.t. every node in the path other than $X$ & $Y$ is **post-dominated** by $Y$
- $X$ is not post-dominated by $Y$

Control dependence graph:
- $Y$ proper descendant of $X$ iff $Y$ control-dependent on $X$
- 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

1. \( y := p + q \)
2. \( x > 0? \)
3. \( a := x \times y \)
4. \( a := y - 2 \)
5. \( w := y / q \)
6. \( x > 0? \)
7. \( b := l \ll w \)
8. \( r := a \& b \)

An example with a loop

Flow of data captured directly in operand dataflow edges
Also have control flow edges sequencing these nodes
- or some more refined control dependence edges

Operators as nodes

Before: nodes in CFG were simple assignments
  - could have operations on r.h.s.
  - used variable names to refer to other values

Alternative: treat the operators themselves as the nodes
  - refer directly other other nodes for their operands

Example

\( p := \& r; \)
\( x := \text{ } * p; \)
\( a := x \times y; \)
\( w := x; \)
\( x := a + a; \)
\( v := y \times w; \)
\( a := v \times 2; \)
Improvements

Bypass variable stores and loads
- i.e., build def/use chains

Treat variable names as (temporary) labels on nodes
- a variable reference implemented by an edge from the node with that label
- a variable assignment shifts the label

The nodes themselves become
the subscripted variables of SSA form

Each computation has its own name (i.e., itself)

More improvements

“Value numbering”:
merge all nodes that compute the same result
- same operator
- same data operands (recursively)
- same control dependence conditions
- operator is pure

Implements (local) CSE

Can do this bottom-up as nodes are initially constructed
- “hash consing”
In face of possibly cyclic data dependence edges,
an optimistic algorithm can get better results [Alpern et al. 88]

Would like to support algebraic identities, too, e.g.
- commutative operators
- \( x \times x = x^2 \)
- associativity, distributivity

Another example

```plaintext
y := p + q;
if m > 1 then
  a := y \times x;
b := a;
else
  b := x - 2;
a := b;
endif
if m < 1 then
d := y \times x;
else
d := x - 2;
endif
w := a / x;
u := b / x;
t := d / x;
if m > 1 then
c := y \times x;
else
c := x - 2;
endif
z := c / x;
```

The example, in SSA form

```plaintext
y := p + q;
if m > 1 then
  a_1 := y \times x; b_1 := a_1;
else
  b_2 := x - 2; a_2 := b_2;
  a_3 := o(a_1, a_2);
b_3 := o(b_1, b_2);
  if m < 1 then
    d_1 := y \times x;
  else
    d_2 := x - 2;
  endif
  d_3 := o(d_1, d_2);
w := a_3 / x;
u := b_3 / x;
t := d_3 / x;
  if m > 1 then
    c_1 := y \times x;
  else
    c_2 := x - 2;
  endif
  c_3 := o(c_1, c_2);
z := c_3 / x;
```
**An improvement**

φ-functions were treated poorly

- impure, since don’t know when they’re the same
  - even if they have the same operands
    and are in the same control equivalent region!

Fix: give them an additional input: the selector value
  (now called select nodes, sometimes written as γ)

- e.g., a boolean, for a 2-input φ
- e.g., an integer, for an n-input φ

φ-functions now are pure!

**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
  ⇒ demand-driven evaluation model

Loops implemented as tail-recursive calls to local procedures

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

**Example, after store splitting**

\[
\begin{align*}
  y &:= p + q; \\
  \text{if } x > 0 \text{ then } a &:= x \times y \text{ else } a := y - 2; \\
  w &:= y / q; \\
  \text{if } x > 0 \text{ then } b &:= 1 << w; \\
  x &:= a \% b;
\end{align*}
\]

**Sequentialization**

How to generate code from a soup of operators and edges?

Need to sequentialize back into a regular CFG

Must find an ordering that respects dependences (data and control)

Hard with arbitrary graph

- can get cycles with full PDG, VDG transforms
- may need to duplicate code to get a legal schedule
Sequentialization via placement

A solution, due to Click: treat as placement problem
- limits transformations/optimizations possible
+ simpler to implement

Start from original (empty) CFG

Goal: assign each operation to
the least-frequently-executed basic block
that respects its data dependences
• φ-nodes tied to their original merge point

Hoist operations out of loops where possible
Push operations into conditionals where possible

Example

i := 0;
while ... do
 x := i * b;
 if ... then
  w := c * c;
  y := x + w;
 else
  y := 9;
 end
 print(y);
 i := i + 1;
end

Example, in SSA form

i_1 := 0;
while ... do
 i_3 := φ(i_1, i_2);
 x := i_3 * b;
 if ... then
  w := c * c;
  y_1 := x + w;
 else
  y_2 := 9;
 end
 y_3 := φ(y_1, y_2);
 print(y_3);
 i_2 := i_3 + 1;
end