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 φ-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 ⇒ 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 * y
4  a := y - 2
5  w := y / q
6  x > 0?
7  b := 1 << w
8  r := a % b
```

An example with a loop

```
B1 -> B2 -> B3
|       |       |
|       |       |
|  T    |  F    |

B1 -> B4 -> B5 -> B6
|       |       |
|       |       |
|  T    |  F    |

B1 -> B7
```
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

Node ::= Constant // 0 operands
   | Var   // 0 operands
   | &Var  // 0 operands
   | Unop  // 1 operand
   | Binop // 2 operands
   | *(ptr deref) // 1 operand
   | . (field deref) // 1 operand
   | [] (array deref) // 2 operands
   | φ // n operands
   | Fn() // n operands
   | Var:= (var assn) // 1 operand
   | *:= (ptr assn) // 2 operands

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

Example

p := &r;
x := *p;
a := x * y;
w := x;
x := a + a;
v := y * w;
a := v * 2;

An improvement

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)

Another improvement

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

Implements (local) CSE

Can do this bottom-up as nodes are initially constructed
- "hash cons'ing"
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
  - $x + x = x * 2$
- associativity, distributivity
Another example

\[ y := p + q; \]
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
\[ w := a / r; \]
\[ u := b / r; \]
\[ t := d / r; \]
if \( m > 1 \) then
\[ c := y \times x; \]
else
\[ c := x - 2; \]
endif
\[ z := c / r; \]

The example, in SSA form

\[ y := p + q; \]
if \( m > 1 \) then
\[ a_1 := y \times x; \]
b_1 := a;
else
\[ b_2 := x - 2; \]
a_2 := b_2;
\[ a_3 := \phi(a_1,a_2); \]
b_3 := \phi(b_1,b_2);
if \( m < 1 \) then
\[ d_1 := y \times x; \]
else
\[ d_2 := x - 2; \]
d_3 := \phi(d_1,d_2);
\[ w := a_3 / r; \]
u := b_3 / r;
t := d_3 / r;
if \( m > 1 \) then
\[ c_1 := y \times x; \]
else
\[ c_3 := x - 2; \]
c_3 := \phi(c_1,c_2);
z := c_3 / r;

An improvement

\( \phi \)-functions are 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 \( \gamma \))
- e.g., a boolean, for a 2-input \( \phi \)
- e.g., an integer for an \( n \)-input \( \phi \)

\( \phi \)-functions now are pure functions!

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
  \( \Rightarrow \) 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
  \( \Rightarrow \) a select node, taking test, then, and else inputs
  \( \Rightarrow \) demand-driven evaluation model

Loops implemented as tail-recursive calls to local procedures

Apply CSE, folding, etc. as nodes are built/updated
**VDG for example, after store splitting**

y := p + q;
if x > 0 then a := x * y else a := y - 2;
w := y / q;
if x > 0 then b := 1 << w;
r := a % b;

---

**Sequentialization**

How to generate code for a soup of operator nodes?

- need to sequentialize back into regular CFG

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

Click’s representation: keeps original CFG around as a guide

- limits transformations/optimizations possible
+ turns sequentialization problem into simpler placement problem

---

**Placement**

Goal: assign each operation to the least-frequently-executed basic block that respects its data dependences

- φ-nodes tied to their original basic block

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
  \[
i_3 := \phi(i_1, i_2);
\]
  \[
x := i_3 * b;
\]
  \[
  \text{if} \ldots \text{then}
  \]
  \[
  w := c * c;
  \]
  \[
y_1 := x + w;
\]
  \[
  \text{else}
  \]
  \[
y_2 := 9;
  \]
  \[
  \text{end}
  \]
  \[
  y_3 := \phi(y_1, y_2);
  \]
  \[
  \text{print}(y_3);
  \]
  \[
  i_2 := i_3 + 1;
  \]
end