Program Representations

Representing programs

- **Goals**
  - analysis is easy and effective
  - just a few cases to handle
  - direct link related things
  - transformations are easy to perform
  - general, across input languages and target machines

- **Additional goals**
  - compact in memory
  - easy to translate to and from
  - tracks info from source through to binary, for source-level debugging, profiling, typed binaries
  - extensible (new options, targets, language features)
  - displayable

Option 1: high-level syntax based IR

- Represent source-level structures and expressions directly
- Example: Abstract Syntax Tree

```
for i := 1 to 10 do
  a[i] := b[i] * 5;
end
```

Option 2: low-level IR

- Translate input programs into low-level primitive chunks, often close to the target machine
- Examples: assembly code, virtual machine code (e.g. stack machines), three-address code, register-transfer language (RTL)

<table>
<thead>
<tr>
<th>Standard RTL instrs:</th>
</tr>
</thead>
<tbody>
<tr>
<td>assignment x := y;</td>
</tr>
<tr>
<td>unary op x := op y;</td>
</tr>
<tr>
<td>binary op x := y op z;</td>
</tr>
<tr>
<td>load x := *(p + o);</td>
</tr>
<tr>
<td>store *(p + o) := x;</td>
</tr>
<tr>
<td>call x := f(...)();</td>
</tr>
<tr>
<td>unary compare op x ?</td>
</tr>
<tr>
<td>binary compare x op y ?</td>
</tr>
</tbody>
</table>

Option 2: low-level IR

```
for i := 1 to 10 do
  a[i] := b[i] * 5;
end
```

Control flow graph containing RTL instructions:
Comparison

• Advantages of high-level rep
  – analysis can exploit high-level knowledge of constructs
  – easy to map to source code (debugging, profiling)
• Advantages of low-level rep
  – can do low-level, machine specific reasoning
  – can be language-independent
• Can mix multiple reps in the same compiler

Components of representation

• Control dependencies: sequencing of operations
  – evaluation of if & then
  – side-effects of statements occur in right order
• Data dependencies: flow of definitions from defs to uses
  – operands computed before operations
• Ideal: represent just dependencies that matter
  – dependencies constrain transformations
  – fewest dependences ⇒ flexibility in implementation

Control dependencies

• Option 1: high-level representation
  – control implicit in semantics of AST nodes
• Option 2: control flow graph (CFG)
  – nodes are individual instructions
  – edges represent control flow between instructions
• Options 2b: CFG with basic blocks
  – basic block: sequence of instructions that don’t have any branches, and that have a single entry point
  – BB can make analysis more efficient: compute flow functions for an entire BB before start of analysis

Data dependencies

• Simplest way to represent data dependencies: def/use chains

Control dependencies

• CFG does not capture loops very well
  – Some fancier options include:
    – the Control Dependence Graph
    – the Program Dependence Graph
• More on this later. Let’s first look at data dependencies
Def/use chains

- Directly captures dataflow
  - works well for things like constant prop
- But...
- Ignores control flow
  - misses some opt opportunities since conservatively considers all paths
  - not executable by itself (for example, need to keep CFG around)
  - not appropriate for code motion transformations
- Must update after each transformation
- Space consuming

SSA

- Static Single Assignment
  - invariant: each use of a variable has only one def

SSA

- Create a new variable for each def
- Insert \( \phi \) pseudo-assignments at merge points
- Adjust uses to refer to appropriate new names
- Question: how can one figure out where to insert \( \phi \) nodes using a liveness analysis and a reaching defns analysis.

Converting back from SSA

- Semantics of \( x_3 := \phi(x_1, x_2) \)
  - set \( x_3 \) to \( x_i \) if execution came from \( i \)th predecessor
- How to implement \( \phi \) nodes?

Converting back from SSA

- Semantics of \( x_3 := \phi(x_1, x_2) \)
  - set \( x_3 \) to \( x_i \) if execution came from \( i \)th predecessor
- How to implement \( \phi \) nodes?
  - Insert assignment \( x_3 := x_1 \) along 1st predecessor
  - Insert assignment \( x_3 := x_2 \) along 2nd predecessor
- If register allocator assigns \( x_1 \), \( x_2 \) and \( x_3 \) to the same register, these moves can be removed
  - \( x_1 \ldots x_n \) usually have non-overlapping lifetimes, so this kind of register assignment is legal
**Recall: Common Sub-expression Elim**

- Want to compute when an expression is available in a var
- Domain:

\[ \{ x \to E, \gamma \to E_\gamma, \zeta \to E_\zeta \} \]

**Recall: CSE Flow functions**

\[ F_{x := Y \text{ op } Z}(\text{in}) = \text{in} \setminus \{ X \to ^* \} \]

\[ \{ \text{out} \} \cup \{ X \to Y \text{ op } Z \mid X = Y \land X \neq Z \} \]

\[ F_{x := Y}(\text{in}) = \text{in} \setminus \{ X \to ^* \} \]

\[ \{ \text{out} \} \cup \{ X \to E \mid Y \to E \in \text{in} \} \]

**Example**

\[ i := a + b \]
\[ x := i \times 4 \]
\[ j := i \]
\[ i := c \]
\[ z := j \times 4 \]

\[ m := b + a \]
\[ w := 4 \times m \]

**Example**

\[ i := a + b \]
\[ x := i \times 4 \]
\[ y := i \times 4 \]
\[ i := i + 1 \]
\[ m := b + a \]
\[ w := 4 \times m \]

**Example**

\[ i := a + b \]
\[ x := i \times 4 \]
\[ j := i \]
\[ i := c \]
\[ z := j \times 4 \]

\[ m := b + a \]
\[ w := 4 \times m \]

**Problems**

- \[ z := j \times 4 \] is not optimized to \[ z := x \], even though \[ x \] contains the value \[ j \times 4 \]
- \[ m := b + a \] is not optimized, even though \[ a + b \] was already computed
- \[ w := 4 \times m \] it not optimized to \[ w := x \], even though \[ x \] contains the value \[ 4 \times m \]

**Problems: more abstractly**

- Available expressions overly sensitive to name choices, operand orderings, renamings, assignments
- Use SSA: distinct values have distinct names
- Do copy prop before running available exprs
- Adopt canonical form for commutative ops
Example in SSA

\[
\begin{align*}
X := Y \text{ op } Z \\
in &\quad F_{X := Y \text{ op } Z}(in) = \\
\text{out} &\quad F_{X := \phi(Y,Z)}(in_0, in_1) =
\end{align*}
\]

Example in SSA

\[
\begin{align*}
X := Y \text{ op } Z \\
in &\quad F_{X := Y \text{ op } Z}(in) = \{ X \rightarrow Y \text{ op } Z \}
\end{align*}
\]

\[
\begin{align*}
X := \phi(Y,Z) \\
in &\quad F_{X := \phi(Y,Z)}(in_0, in_1) = (in_0 \cap in_1) \cup \\
&\quad \{ X \rightarrow E \mid Y \rightarrow E \in in_0 \land Z \rightarrow E \in in_1 \}
\end{align*}
\]

Example in SSA

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

Example in SSA

\[
\begin{align*}
\hat{i}_1 := a_1 + b_1 \\
x_1 := \hat{i}_1 \times 4 \\
\hat{j}_1 := \hat{i}_1 \\
\hat{i}_2 := c_1 \\
x_2 := \hat{j}_1 \times 4 \\
i_1 := b_1 \\
w_1 := a_1 \times 4
\end{align*}
\]

What about pointers?

- Pointers complicate SSA. Several options.
  - Option 1: don’t use SSA for pointed to variables
  - Option 2: adapt SSA to account for pointers
  - Option 3: define src language so that variables cannot be pointed to (eg: Java)

SSA helps us with CSE

- Let’s see what else SSA can help us with
  - Loop-invariant code motion
## Loop-invariant code motion

- Two steps: analysis and transformations

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

- Step 2: move them outside loop
  - to top if used within loop: code hoisting
  - to bottom if used after loop: code sinking

### Example

```
x := 3
p := w + y
x := x + 1
q := q + 1
z := x * y
q := y * y
w := y + 2
y := 4
y := 5
w := w + 5
```

### Detecting loop invariants

- An expression is invariant in a loop L iff:
  - (base cases)
    - it's a constant
    - it's a variable use, all of whose defs are outside of 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 invariants

- Option 1: iterative dataflow analysis
  - optimistically assume all expressions loop-invariant, and propagate

- Option 2: build def/use chains
  - follow chains to identify and propagate invariant expressions

- Option 3: SSA
  - like option 2, but using SSA instead of def/use chains

### Example using def/use chains

```
x := 3
y := 4
y := 5
x := x + 1
q := q + 1
```

- An expression is invariant in a loop L iff:
  - (base cases)
    - it's a constant
    - it's a variable use, all of whose defs are outside of 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
Example using def/use chains

• An expression is invariant in a loop L iff:
  (base cases)
  – it's a constant
  – it's a variable use, all of whose defs are outside of 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

Loop invariant detection using SSA

• An expression is invariant in a loop L iff:
  (base cases)
  – it's a constant
  – it's a variable use, all of whose single defs are outside of L
  (inductive cases)
  – it's a pure computation all of whose args are loop-invariant
  – it's a variable use whose single reaching def, and the rhs of that def is loop-invariant
  • $\phi$ functions are not pure

Example using SSA

• An expression is invariant in a loop L iff:
  (base cases)
  – it's a constant
  – it's a variable use, all of whose single defs are outside of L
  (inductive cases)
  – it's a pure computation all of whose args are loop-invariant
  – it's a variable use whose single reaching def, and the rhs of that def is loop-invariant
  • $\phi$ functions are not pure

Example using SSA and preheader

• An expression is invariant in a loop L iff:
  (base cases)
  – it's a constant
  – it's a variable use, all of whose single defs are outside of L
  (inductive cases)
  – it's a pure computation all of whose args are loop-invariant
  – it's a variable use whose single reaching def, and the rhs of that def is loop-invariant
  • $\phi$ functions are not pure

Summary: Loop-invariant code motion

• Two steps: analysis and transformations
  • Step1: find invariant computations in loop
    – invariant: computes same result each time evaluated
  • Step 2: move them outside loop
    – to top if used within loop: code hoisting
    – to bottom if used after loop: code sinking

Code motion

• Say we found an invariant computation, and we want to move it out of the loop (to loop pre-header)
  • When is it legal?
  • Need to preserve relative order of invariant computations to preserve data flow among move statements
• Need to preserve relative order between invariant computations and other computations
Example

```
x := 0
y := 1
i := 0
```

```
x := a * b
y := x / z
i := i + 1
```

```
x := 0
y := 1
i := 0
```

Lesson from example: domination restriction

- To move statement S to loop pre-header, S must \textbf{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 constraint at cost of possibly slowing down the program

Domination restriction in for loops

Avoiding domination restriction

- Domination restriction strict
  - Nothing inside branch can be moved
  - Nothing after a loop exit can be moved

- Can be circumvented through loop normalization
  - while-do $\Rightarrow$ if-do-while

Another example

```
x := 5
i := 0
```

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

```
x := 0
```

```
i := i + 1
```

```
i < N ?
```

```
\ldots \& \ldots
```
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 reached by any def other than $S$

• Otherwise may reorder defs/uses

Avoiding data restriction

Avoiding data restriction

Summary of Data dependencies

• We’ve seen SSA, a way to encode data dependencies better than just def/use chains
  – makes CSE easier
  – makes loop invariant detection easier
  – makes code motion easier

• Now we move on to looking at how to encode control dependencies

Control Dependencies

• A node (basic block) $Y$ is control-dependent on another $X$ iff $X$ determines whether $Y$ executes
  – there exists a path from $X$ to $Y$ s.t. every node in the path other than $X$ and $Y$ is post-dominated by $Y$
  – $X$ is not post-dominated by $Y$
**Control Dependence Graph**

- Control dependence graph: Y descendent of X iff Y is control dependent on X
  - label each child edge with required condition
  - group all children with same condition under region node
- Program dependence graph: super-impose dataflow graph (in SSA form or not) on top of the control dependence graph
Another example

Summary of Control Dependence Graph

- More flexible way of representing control-depencies than CFG (less constraining)
- Makes code motion a local transformation
- However, much harder to convert back to an executable form

Course summary so far

- Dataflow analysis
  - flow functions, lattice theoretic framework, optimistic iterative analysis, precision, MOP
- Advanced Program Representations
  - SSA, CDG, PDG
- Along the way, several analyses and opts
  - reaching defns, const prop & folding, available exprs & CSE, liveness & DAE, loop invariant code motion
- Pointer analysis
  - Andersen, Steensgaard, and long the way: flow-insensitive analysis
- Next: dealing with procedures