Program Representations
Representing programs

• Goals
Representing programs

• Primary goals
  – analysis is easy and effective
    • just a few cases to handle
    • directly 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, profilling, typed binaries
  – extensible (new opts, targets, language features)
  – displayable
Option 1: high-level syntax based IR

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

Source:

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

AST:
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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>assignment</td>
<td>( x := y; )</td>
</tr>
<tr>
<td>unary op</td>
<td>( x := \text{op} \ y; )</td>
</tr>
<tr>
<td>binary op</td>
<td>( x := y \ \text{op} \ z; )</td>
</tr>
<tr>
<td>address-of</td>
<td>( p := &amp; y; )</td>
</tr>
<tr>
<td>load</td>
<td>( x := *(p + o); )</td>
</tr>
<tr>
<td>store</td>
<td>( *(p + o) := x; )</td>
</tr>
<tr>
<td>call</td>
<td>( x := f(\ldots); )</td>
</tr>
<tr>
<td>unary compare</td>
<td>( \text{op} \ x \ ? )</td>
</tr>
<tr>
<td>binary compare</td>
<td>( x \ \text{op} \ y \ ? )</td>
</tr>
</tbody>
</table>
Option 2: low-level IR

Source:

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

Control flow graph containing RTL instructions:

1. `i := 1`
2. `i <= 10?`
3. `t1 := i * 4`
4. `t2 := & b`
5. `t3 := *(t2 + t1)`
6. `t4 := t3 * 5`
7. `t5 := i * 4`
8. `t6 := & a`
9. `*(t6 + t5) := t4`
10. `i := i + 1`
Comparison
## Comparison

<table>
<thead>
<tr>
<th>Advantages of high-level rep</th>
<th>Advantages of low-level rep</th>
<th>Can mix multiple reps in the same compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>- analysis can exploit high-level knowledge of constructs</td>
<td>- can do low-level, machine specific reasoning</td>
<td></td>
</tr>
<tr>
<td>- easy to map to source code (debugging, profiling)</td>
<td>- can be language-independent</td>
<td></td>
</tr>
</tbody>
</table>
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 $\Rightarrow$ 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
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
Data dependencies

• Simplest way to represent data dependencies: def/use chains
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
• Static Single Assignment
  – invariant: each use of a variable has only one def
\[
\begin{align*}
  i_1 &:= 0 \\
i_2 &:= \phi(i_1, i_2) \\
i_3 &:= i_2 + 1
\end{align*}
\]

\[
\begin{align*}
x_1 &:= \ldots \\
y_1 &:= \ldots \\
x_2 &:= x_1 + y_1 \\
\ldots &\ldots x_2 \ldots
\end{align*}
\]

\[
\begin{align*}
x_3 &:= \ldots \\
y_2 &:= y_1 + 1 \\
x_4 &:= \ldots x_3 \ldots
\end{align*}
\]

\[
\begin{align*}
y_3 &:= \phi(y_1, y_2) \\
x_4 &:= \phi(x_2, x_3)
\end{align*}
\]

\[
\begin{align*}
\ldots &\ldots y_3 \ldots \\
\ldots &\ldots y \ldots
\end{align*}
\]
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 $1^{st}$ predecessor
  – Insert assignment $x_3 := x_2$ along $2^{nd}$ predecessor

• If register allocator assigns $x_1$, $x_2$ and $x_3$ to the same register, these moves can be removed
  – $x_1 .. x_n$ 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:
  \[
  \{ x \rightarrow E_1, \ y \rightarrow E_2, \ z \rightarrow E_3 \} \\
  S = \{ x \rightarrow E \mid x \in \text{Var}, \ E \in \text{Exp} \}
  \]

\[ \emptyset = S \]
\[ t = S \]
\[ T = \emptyset \]
\[ u = \top \]
Recall: CSE Flow functions

\[
F_X := Y \text{ op } Z \text{(in)} = \text{ in } \left\{ X \rightarrow * \right\} \\
\left\{ * \rightarrow \ldots X \ldots \right\} \cup \\
\{ X \rightarrow Y \text{ op } Z \mid X \neq Y \land X \neq Z \}
\]

\[
F_X := Y \text{(in)} = \text{ in } \left\{ X \rightarrow * \right\} \\
\left\{ * \rightarrow \ldots X \ldots \right\} \cup \\
\{ X \rightarrow E \mid Y \rightarrow E \in \text{ in } \}
\]
Example

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

\[
j := i \\
i := c \\
z := j \times 4
\]

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

\[
y := i \times 4 \\
i := i + 1
\]
Example

\[
\begin{align*}
i &:= a + b \\
x &:= i \times 4 \\
i &:= i + 1 \\
m &:= b + a \\
w &:= 4 \times m
\end{align*}
\]

\[
\begin{align*}
j &:= i \\
i &:= c \\
z &:= j \times 4 \\
y &:= i \times 4
\end{align*}
\]
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

\[ X := \mathbf{Y \text{ op } Z} \]

\[ F_X := \mathbf{Y \text{ op } Z}(\text{in}) = \]

\[ X := \phi(\mathbf{Y, Z}) \]

\[ F_X := \phi(\mathbf{Y, Z})(\text{in}_0, \text{in}_1) = \]

\[ \{ X \rightarrow E | \mathbf{Y} \leftarrow E \in \text{in}_0 \land \mathbf{Z} \leftarrow E \in \text{in}_1 \} \]
Example in SSA

\[ X := Y \text{ op } Z \]

\[ F_X := Y \text{ op } Z(\text{in}) = \text{in} \cup \{ X \rightarrow Y \text{ op } Z \} \]

\[ X := \phi(Y, Z) \]

\[ F_X := \phi(Y, Z)(\text{in}_0, \text{in}_1) = (\text{in}_0 \cap \text{in}_1) \cup \{ X \rightarrow E \mid Y \rightarrow E \in \text{in}_0 \wedge Z \rightarrow E \in \text{in}_1 \} \]
Example in SSA

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

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

m := b + a
w := 4 * m

y := i * 4
i := i + 1
```
Example in SSA

\[i_1 := a_1 + b_1\]
\[x_1 := i_1 \times 4\]

\[j_1 := i_1\]
\[i_2 := c_1\]
\[z_1 := i_1 \times 4\]

\[i_4 := \phi(i_1, i_3)\]
\[y_1 := i_4 \times 4\]
\[i_3 := i_4 + 1\]

\[m_1 := a_1 + b_1\]
\[w_1 := m_1 \times 4\]
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

• 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
Example

\[ x := 3 \]
\[ y := 4 \]
\[ y := 5 \]
\[ z := x \times y \]
\[ q := y \times y \]
\[ w := y + 2 \]
\[ w := w + 5 \]
\[ p := w + y \]
\[ x := x + 1 \]
\[ q := q + 1 \]
Example

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

- 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

\[
x := 3
\]
\[
y := 4
\]
\[
y := 5
\]
\[
z := x \cdot y
\]
\[
q := y \cdot y
\]
\[
w := y + 2
\]
\[
w := w + 5
\]
\[
p := w + y
\]
\[
x := x + 1
\]
\[
q := q + 1
\]
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

\[
\begin{align*}
x & := 3 \\
y & := 4 \\
z & := x \times y \\
w & := y + 2 \\
p & := w + y \\
x & := x + 1 \\
q & := q + 1 \\
q & := y \times y \\
w & := w + 5
\end{align*}
\]
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:
  - it’s a constant
  - it’s a variable use, all of whose single defs are outside of L
  - 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

- 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**
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

\[ x := a \times b \]
\[ y := x / z \]
\[ i := i + 1 \]
\[ z \neq 0 \land i < 100 \]
\[ q := x + 1 \]
Lesson from example: domination restriction

- To move statement S to loop pre-header, S must dominate all loop exits
  [ A dominates B when all paths to B 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

\[ i := 0 \]

\[ i < N? \]

\[ x := a / b \]
\[ i := i + 1 \]
Domination restriction in for loops

Before

\[ i := 0 \]

\[ i < N? \]

\[ x := a / b \]

\[ i := i + 1 \]

After

\[ i := 0 \]

\[ i < N? \]

\[ x := a / b \]

\[ i := i + 1 \]

\[ i < N? \]
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 => if-do-while
Another example

```
z := 5
i := 0

z := z + 1

z := 0

i := i + 1

i < N ?

... z ...
```
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

```
z := 5
i := 0

z := z + 1
z := 0
i := i + 1
i < N?

... z ...
Avoiding data restriction

- Restriction unnecessary in SSA!!
- Implementation of phi nodes as moves will cope with re-ordered defs/uses

```
z_1 := 5
i_1 := 0

z_2 := \phi(z_1, z_4)
i_2 := \phi(i_1, i_3)
z_3 := z_2 + 1
z_4 := 0
i_3 := i_2 + 1
i_3 < N ?
```

... z_4 ...

... z_4 ...
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 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
Example

Proc

1) \( y := p + q \)
2) \( x > 0? \)

3) \( a := x \times y \)
4) \( a := y - 2 \)

5) \( w := y / q \)
6) \( x > 0? \)

7) \( b := 1 << w \)

8) \( r := a \% b \)

7 cd0 6
3,4 cd0 2
Example

```
Proc

1 y := p + q
2 if x > 0
3 a := x * y
4 a := y - 2
5 w := y / q
6 if x > 0
7 b := 1 << w
8 r := a % b
```

Control dependence relation:
3 depends on 2
4 depends on 2
7 depends on 6

Proc
1 2 5 6 
3 4 7
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
Example

1. $y := p + q$
2. $x > 0$?

3. $a := x \times y$
4. $a := y - 2$

5. $w := y \div q$
6. $x > 0$?

7. $b := 1 \ll w$

8. $r := a \% b$

Control dependence relation:
3 depends on 2
4 " " 2
7 " " 6
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
```

Control dependence relation:
3 depends on 2
4 depends on 2
7 depends on 6

Root: 1

Directions:
- T: True
- F: False
- R: Root
Another example
Another example
Another example

```plaintext
i_1 := 0;
while
  ... do
  i_3 := \phi(i_1, i_2);
  x := i_3 * b;
  if
    ... then
    w := c * c;
    Y_1 := 9 + w;
  else
    Y_2 := 9;
  end
  Y_3 := \phi(Y_1, Y_2);
  print(y_3);
  i_2 := i_3 + 1;
end
```
Summary of Control Dependency Graph

• More flexible way of representing control-dependencies 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