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, profiling, 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>Instruction</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>assignment</td>
<td>x := y;</td>
</tr>
<tr>
<td>unary op</td>
<td>x := op y;</td>
</tr>
<tr>
<td>binary op</td>
<td>x := y 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(...);</td>
</tr>
<tr>
<td>unary compare</td>
<td>op x ?</td>
</tr>
<tr>
<td>binary compare</td>
<td>x op y ?</td>
</tr>
</tbody>
</table>
Option 2: low-level IR

Source:

\[
\begin{array}{l}
\text{for } i := 1 \text{ to } 10 \text{ do} \\
\quad a[i] := b[i] \times 5; \\
\text{end}
\end{array}
\]

Control flow graph containing RTL instructions:

\[
\begin{align*}
i & := 1 \\
i & \leq 10? \\
t1 & := i \times 4 \\
t2 & := b \\
t3 & := (t2 + t1) \\
t4 & := t3 \times 5 \\
t5 & := i \times 4 \\
t6 & := a \\
*(t6 + t5) & := t4 \\
i & := i + 1
\end{align*}
\]
Comparison
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

\[
\begin{align*}
X &:= 5 \\
Y &:= 10
\end{align*}
\]
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
SSA

• Static Single Assignment
  – invariant: each use of a variable has only one def
\(x := \ldots\)
\(y := \ldots\)
\(x := x + y\)
\(y := y + 1\)
\(y_3 = \varphi(y, y_2)\)
\(x_4 = \varphi(x_3, y_3)\)
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\textsuperscript{st} predecessor
  – Insert assignment $x_3 := x_2$ along 2\textsuperscript{nd} 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:

\[ \{ x \mapsto E_1, \ y \mapsto E_2, \ z \mapsto E_3 \} \]

\[ S = \{ x \mapsto E \mid x \in \text{Var}, \ E \in \text{Exp} \} \]

\[ S = \{ x \mapsto E \mid x \in \text{Var}, \ E \in \text{Exp} \} \]

- \( \emptyset \in S \)
- \( x \in S \)
- \( \top \in S \)
- \( \bot \in \emptyset \)
Recall: CSE Flow functions

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

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

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

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

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

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

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

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

• $w := 4 \times m$ is 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

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

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

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

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

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

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

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

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

\[
\begin{align*}
i_1 &:= a + b \\
x_1 &:= i_1 \cdot 4 \\
j_1 &:= i_1 \\
i_2 &:= c \\
z_1 &:= j_1 \cdot 4 \\
m_1 &:= b + a \\
w_1 &:= 4 \cdot m_1
\end{align*}
\]
Example in SSA

\[ x := x d \]

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

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

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

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

- 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

\[
x := 3
\]

\[
y := 4
\]

\[
z := x \times y
\]

\[
p := w + y
\]

\[
x := x + 1
\]

\[
q := q + 1
\]

\[
y := 5
\]

\[
q := y \times y
\]

\[
w := y + 2
\]

\[
w := w + 5
\]
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 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
p := w + y
z := x * y
q := y * y
w := y + 2

w := w + 5
x := x + 1
q := q + 1

y := 5
```
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

```
x_1 := 3
y_1 := 4
y_2 := 5
x_2 := \phi(x_1, x_3)
y_3 := \phi(y_1, y_2, y_3)
z_1 := x_2 * y_3
q_1 := y_3 * 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
```
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

\[ x := a \times b \]
\[ y := x / z \]
\[ i := i + 1 \]

\[ t := \text{read}(\ldots) \]
\[ x := 0 \]
\[ y := 1 \]
\[ i := 0 \]

\[ \text{if } (*) \]

\[ q := x + 1 \]
Example

\[
x := a \times b \\
y := x / z \\
i := i + 1
\]

\[
x := 0 \\
y := 1 \\
i := 0
\]

\[
z \neq 0 \&\& \\
i < 100 ?
\]

\[
q := x + 1
\]
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

1 < N?
```
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

\[ z_{1} := 5 \]
\[ i_{1} := 0 \]
\[ z_{3} := 0 \]
\[ z_{4} := \varphi (z_{1}, z_{3}) \]
\[ i_{3} := \varphi (i_{1}, i_{2}) \]
\[ z_{2} := z_{4} + 1 \]
\[ z_{3} := 0 \]
\[ i_{2} := i_{2} + 1 \]
\[ i_{2} < N \]

\[ \ldots z_{3} \ldots \]
Avoiding data restriction

\[ z_1 := 5 \]
\[ i_1 := 0 \]

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

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

```
Call

= Entry

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

X is cp < y
3 depends 2
4 " " "
7 " " 6
1 " 0
2 | 0
5 | 0
8 " 0
```
Example

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

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 \( b := 1 \ll w \)

8 \( r := a \% b \)
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 * 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 " " 2
7 " " 6
Example

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

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
Another example
Another example
Another example

\begin{enumerate}
\item $i_1 := 0$
\item while \ldots do
\item $i_3 := \phi(i_1, i_2)$
\item $x := i_3 \times b$
\item if \ldots then
\item $w := c \times c$
\item $Y_1 := 9 + w$
\item else
\item $Y_2 := 9$
\item end
\item $Y_3 := \phi(Y_1, Y_2)$
\item print($y_3$)
\item $i_2 := i_3 + 1$
\item end
\end{enumerate}
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