Representation of programs

Primary goals:
- analysis is easy & effective
- just a few cases to handle
- provide support for linking things of interest
- transformations are easy
- general, across input languages & target machines

Additional goals:
- compact in memory
- easy to translate to and from
- tracks info for source-level debugging, profiling, etc.
- extensible (new optimizations, targets, language features)
- displayable

High-level syntax-based representation

Represent source-level control structures & expressions directly

Examples
- (Attributed) AST
- Lisp S-expressions
- extended lambda calculus

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

AST:

Low-level representation

Translate input programs into low-level primitive chunks, often close to the target machine

Examples
- assembly code, virtual machine code (e.g. stack machine)
- three address code, register transfer language (RTLs)

Standard RTL operators:

<table>
<thead>
<tr>
<th>Standard RTL operators</th>
<th>Examples</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>
Comparison

Advantages of high-level rep:
- analysis can exploit high-level knowledge of constructs
- probably faster to analyze
- easy to map to source code terms for debugging, profiling
- may be more compact

Advantages of low-level rep:
- can do low-level, machine-specific optimizations
  (if target-based representation)
- high-level rep may not be able to express some transformations
- can have relatively few kinds of instructions to analyze
- can be language-independent

High-level rep suitable for a source-to-source or special-purpose optimizer, e.g. inliner, parallelizer

Can mix multiple representations in single compiler
Can sequence compilers using different reps

Q: what about Java bytecodes?

Representing control dependences

Option 1: high-level representation
- control flow implicit in semantics of AST nodes

Option 2: control flow graph
- nodes are basic blocks
- instructions in basic block sequence side-effects
- edges represent branches (control flow between basic blocks)

Some fancier options:
- control dependence graph, part of program dependence graph (PDG) [Ferrante et al. 87]
- convert into data dependences on a memory state, in value dependence graph (VDG) [Weise et al. 94]

Representing data dependences

Option 1: implicitly through variable defs/uses in CFG
  + simple, source-like
  - may overconstrain order of operations
  - analysis wants important things explicit ⇒ analysis can be slow

Option 2: def/use chains, linking each def to each use
  + explicit ⇒ analysis can be fast
  - must be computed, maintained after transformations
  - may be space-consuming

Some fancier options:
- static single assignment (SSA) form [Alpern et al. 88]
- value dependence graphs (VDGs)
- ...
Data flow analysis

Want to compute some info about program
• at program points
• to identify opportunities for improving transformations

Can model data flow analysis as solving system of constraints
• each node in CFG imposes a constraint relating info at predecessor and successor points
• solution to constraints is result of analysis

Solution must be safe/sound
Solution can be conservative

Key issues:
• how to represent info efficiently?
• how to represent & solve constraints efficiently?
• how long does constraint solving take? does it terminate?
• what if multiple solutions are possible?
• how to synchronize transformations with analysis?
• how to know if analysis & transformations we’ve defined are semantics-preserving?

Example: reaching definitions

For each program point, want to compute set of definitions (statements) that may reach that point
• reach: are the last definition of some variable

Info = set of var->stmt bindings
E.g.:
{x→s1, y→s5, y→s8}

Can use reaching definition info to:
• build def-use chains
• do constant & copy propagation
• detect references to undefined variables
• present use/def info to programmer
• ...

Safety rule (for these intended uses of this info):
can have more bindings than the “true” answer, but can’t miss any

Constraints for reaching definitions

Main constraints:

A simple assignment removes any old reaching defs for the lhs and replaces them with this stmt:
• strong update
  \[ s: x := \ldots \]
  \[ \text{info}_{\text{succ}} = \text{info}_{\text{pred}} - \{x \rightarrow s' \mid s \} \cup \{x \rightarrow s\} \]

A pointer assignment may modify anything, but doesn’t definitely replace anything
• weak update
  \[ s: \ast p := \ldots \]
  \[ \text{info}_{\text{succ}} = \text{info}_{\text{pred}} \cup \{x \rightarrow s \mid \forall x \in \text{may-point-to}(p)\} \]

Other statements: do nothing
\[ \text{info}_{\text{succ}} = \text{info}_{\text{pred}} \]
Constraints for reaching definitions, continued

Branches pass through reaching defs to both successors
\[ \text{info}_{\text{succ}[i]} = \text{info}_{\text{pred}[i]} \quad \forall i \]

Merges take the union of all incoming reaching defs
- we don't know which path is being taken at run-time
  ⇒ be conservative
\[ \text{info}_{\text{succ}} = \bigcup_i \text{info}_{\text{pred}[i]} \]

Conditions at entry to CFG: definitions of formals
\[ \text{info}_{\text{entry}} = \{x \rightarrow \text{entry} \mid \forall x \in \text{formals} \} \]

Solving constraints

A given program yields a system of constraints
Need to solve constraints

For reaching definitions,
can traverse instructions in forward topological order,
computing successor info from predecessor info
- because of how the constraints are defined

Example

1 \(x := \ldots\)
2 \(y := \ldots\)
3 \(y := \ldots\)
4 \(p := \ldots\)

5 \(x := \ldots\)
   \(y \ldots\)

6 \(x := \ldots\)
   \(p := \ldots\)

7 \(x := \ldots\)

8 \(y := \ldots\)

Another example

1 \(x := \ldots\)
2 \(y := \ldots\)
3 \(y := \ldots\)
4 \(p := \ldots\)

5 \(x := \ldots\)

6 \(x := \ldots\)

7 \(p := \ldots\)

8 \(y := \ldots\)

Topological order not defined!
Loop terminology

-loop: strongly-connected component in CFG with single entry

-loop entry edge: source not in loop, target in loop
-loop exit edge: the reverse
-back edge: target is loop head node

-loop head node: target of loop entry edge
-loop tail node: source of back edge
-loop preheader node: single node that’s source of loop entry edge

-nested loop: loop whose head is inside another loop

-reducible flow graph: all SCC’s have single entry

Analysis of loops

If CFG has a loop, data flow constraints are recursively defined:

info_{loop-head} = info_{loop-entry} \cup info_{back-edge}
info_{back-edge} = \ldots info_{loop-head}

Substituting definition of info_{back-edge}:

info_{loop-head} = info_{loop-entry} \cup (\ldots info_{loop-head} \ldots)

Summarizing r.h.s. as \( F \):

info_{loop-head} = F(info_{loop-head})

A legal solution to constraints is a \textbf{fixed-point} of \( F \)

Recursive constraints can have many solutions

- want least or greatest fixed-point, whichever corresponds to the most precise answer

How to find least/greatest fixed-point of \( F \)?

- for restricted CFGs can use specialized methods
- e.g. interval analysis for reducible CFGs
- for arbitrary CFGs, can use iterative approximation

Iterative data flow analysis

1. Start with initial guess of info at loop head:

\[
info_{loop-head} = \text{guess}
\]

2. Solve equations for loop body:

\[
\begin{align*}
info_{back-edge} &= F_{body}(info_{loop-head}) \\
info_{loop-head} &= info_{loop-entry} \cup info_{back-edge}
\end{align*}
\]

3. Test if found fixed-point:

\[
info_{loop-head}' = info_{loop-head} \quad ?
\]

A. if same, then done

B. if not, then adopt result as (better) guess and repeat:

\[
\begin{align*}
info_{back-edge}' &= F_{body}(info_{loop-head}') \\
info_{loop-head}'' &= info_{loop-entry} \cup info_{back-edge}' \\
info_{loop-head}''' &= info_{loop-head}'' \quad ?
\end{align*}
\]

...
When does iterating work?

1. need to be able to make an initial guess
2. info^{n+1} must be closer to the fixed-point than info^n (true if F_{body} is monotonic)
3. must eventually reach the fixed-point in a finite number of iterations (true if info drawn from a finite-height domain)

To reach best fixed-point, initial guess for loop head should be optimistic

- easy choice: info_{loop-head} = info_{loop-entry}

(Even if guess is overly optimistic, iteration will ensure we won’t stop analysis until the answer is safe.)

To speed iterative analysis, want to test guess ASAP

- avoid solving constraints outside of loop until fixed-point is reached within loop

The example, again

1 \ x := ...  
2 \ y := ...  
3 \ y := ...  
4 \ p := ...  

\[ \ldots \times \ldots \]

5 \ x := ...  
   \ldots \ y \ldots  

6 \ x := ...  
   \times \ldots  

7 \ p := ...  

\[ \ldots \times \ldots \]

8 \ y := ...  
   \ldots \ y \ldots  

\[ \ldots \times \ldots \]