Representation of programs in optimizer

Goals:

- analysis is program is easy, direct
- making transformations is easy
- just a few cases to handle
- compact in memory
- easy to generate to and from
- tracks info for source-level debugging, profiling, etc.
- extensible (new optimizations, targets, language features)
- displayable

High-level syntax-based representation

Examples

- AST
- Lisp S-expressions

Represent source-level control structures directly

\+ analysis exploits high-level knowledge of constructs

\+ easy to map to source code terms for debugging, profiling

– some transformations hard

– low-level optimizations hard/impossible

– can be many kinds of AST nodes to analyze

– limited to particular input language(s)

Suitable for a source-to-source optimizer

- rely on another compiler to do low-level optimizations

Low-level target-based representation

Examples

- assembly code, virtual machine code (e.g. stack machine)
- three address code, register transfer language (rtl's):

```
x := y;  x := op y;  x := y op z;
p := &y;  x := *(p + o);  *(p + o) := x;
x := f(...);
op x?   x op y?
```

Translate input programs into low-level primitive chunks

+ can do low-level, machine-specific optimizations
+ supports good analysis, transformations
+ can have relatively few kinds of instructions to analyze
+ can be language-independent

– may lose high-level knowledge if not careful
– hard to track source for debugging, profiling, etc.
– can be more space-, time-consuming

Can have multiple representations at different levels

Components of representation

Operations

Control flow: sequencing of operations

- evaluation of then & else arms depends on result of test
- side-effects of statements in right order

Data flow: flow of values from **definitions** to **uses**

- operands computed before operation

A.k.a. control & data **dependence**

Ideal: represent just those dependences that matter

- dependences constrain transformations
- fewest dependences $\Rightarrow$ most flexibility in implementation

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
- edges represent branches
  (control flow between basic blocks)

Fancier:

- control dependence tree,
  part of program dependence graph (PDG)
- convert into data dependences on explicit state,
  in value dependence graph [Weise *et al.*]

Kinds of data dependences

read-after-write (RAW): true (a.k.a. flow) dependence

- reflects real data flow, operands to operation

write-after-read (WAR): anti-dependence
write-after-write (WAW): output dependence

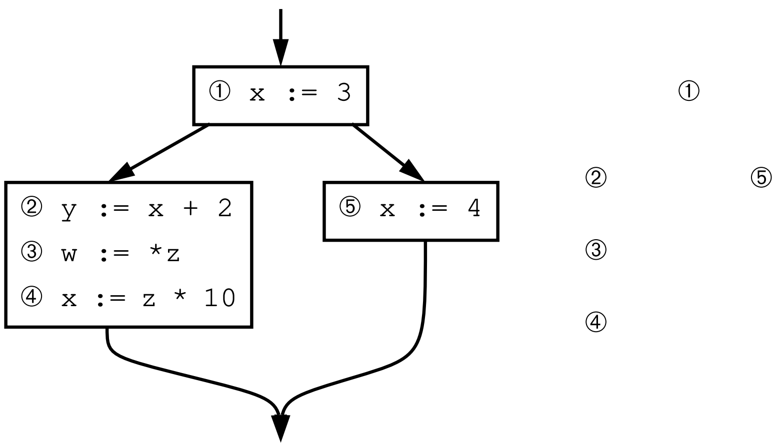
- reflects reuse of memory, not real data flow

read-after-read (RAR): no dependence

- can occur in any order

Example

```
x := 3
if ... then
  y := x + 2
  w := *z
  x := z * 10
else
  x := 4
endif
```



Representing data dependences

Within basic block:

Option 1: sequence of instructions
(represent data flow as a kind of control dependence)

+ simple, source-like
+ fixed ordering supports easy analysis
– may overconstrain order of operations

Option 2: expression tree/DAG

+ natural, abstract
+ directly captures data dependences within basic block
+ DAG supports local CSE
+ can be compact
– conceptually harder to analyze, transform
– must linearize eventually

Representing data dependences, cont.

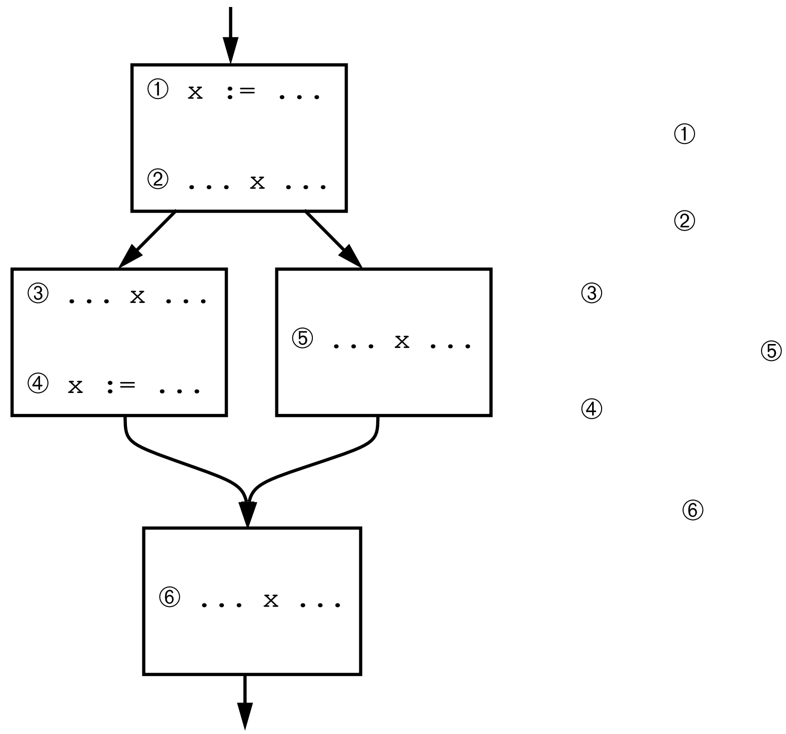
Across basic blocks:

Option 1: implicitly through variable defs/uses
- \+ simple
- – analysis wants important things explicit

Option 2: def/use chains, linking each def with each use
- \+ explicit
- – must be computed, maintained after transformations
- – space consuming

Option 3: static single assignment (SSA) form
Option 4: value dependence graphs
...

Example



Data flow analysis

Want to compute some info about program
- at **program points**
- to determine opportunities for improving transformations

Model info as a set (for now)

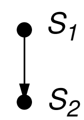
Compute info by setting up and solving **data flow equations**

Data flow equations

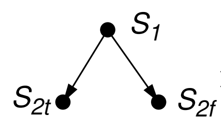
Basic equation for each stmt S:

$\mathtt{OUT}(S) = \mathtt{GEN}(S) \cup (\mathtt{IN}(S) - \mathtt{KILL}(S))$

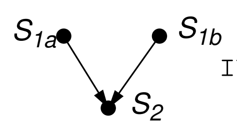
Additional connecting equations model control flow:



$\mathtt{IN}(S_2) = \mathtt{OUT}(S_1)$



$\mathtt{IN}(S_{2t}) = \mathtt{OUT}(S_1)$
$\mathtt{IN}(S_{2f}) = \mathtt{OUT}(S_1)$



$\mathtt{IN}(S_2) = \mathtt{OUT}(S_{1a})$ **merge** $\mathtt{OUT}(S_{1b})$

Set 'em up and solve!

Example: reaching definitions

For each program point,
want to compute set of definitions that *reach* that point

- can use info to build def-use chains, do constant propagation, etc.

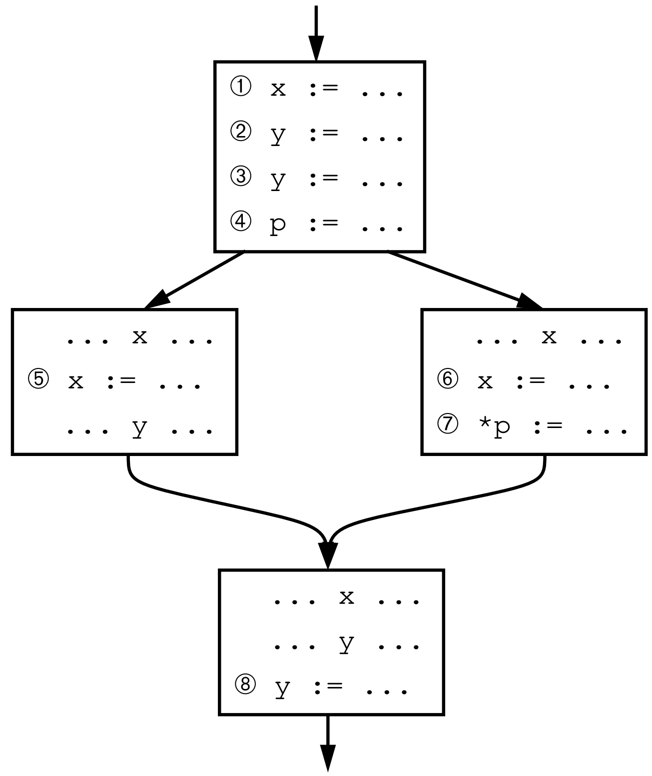
Info = set of (*var* → *rtl*) bindings

GEN(*s*) = {*v* → *s*}, forall *v* s.t. *s* assigns *v*
KILL(*s*) = {*v* → *?*}, forall *v* st. *s* assigns *v*

Initial conditions at start of procedure: {}

- really, special bindings for formal arguments

Example



Conservatism

Problem: what if analysis results are only approximate?

Constraint: don't want to do any illegal transformations

- OK to miss something that could have been legal

Analysis must be **conservative**

- if it is inaccurate, it must err on the side of safety

For reaching definitions:

- include ($v \rightarrow$ *?*) in KILL(*s*) if *s* might modify *v*
- include ($v \rightarrow s$) in GEN(*s*) only if *s* definitely modifies *v*
- at merges, take union of bindings of merging branches

Loops

If CFG has a loop, data flow equations will be recursive:

$$\mathrm{OUT}(loop) = F_{loop}(\mathrm{IN}(loop)\ \textbf{merge}\ \mathrm{OUT}(loop))$$

Want to solve for **fixpoint** of these equations

- want **least**/**greatest** fixpoint

For restricted CFGs can use specialized methods

- **interval analysis** for **reducible** CFGs

For arbitrary CFGs, can use **iterative** approximation

Iterative data flow analysis

1. Start with initial guess of info at end of loop

   $\mathtt{OUT}_0(\mathit{loop})$ = guess

2. Solve equations for loop

   $\mathtt{OUT}_1(\mathit{loop}) = F_{loop}(\mathtt{IN}(\mathit{loop})$ **merge** $\mathtt{OUT}_0(\mathit{loop}))$

3. Test if found fixpoint

   $\mathtt{OUT}_1(\mathit{loop}) = \mathtt{OUT}_0(\mathit{loop})$?

   A. if so, then done

   $\mathtt{OUT}(\mathit{loop}) = \mathtt{OUT}_1(\mathit{loop})$

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

   $\mathtt{OUT}_2(\mathit{loop}) = F_{loop}(\mathtt{IN}(\mathit{loop})$ **merge** $\mathtt{OUT}_1(\mathit{loop}))$

   $\mathtt{OUT}_2(\mathit{loop}) = \mathtt{OUT}_1(\mathit{loop})$?

   ...

When does iterating work?

Need to be able to make an initial guess

$\mathtt{OUT}_{i+1}(\mathit{loop})$ must be closer to the fixpoint than $\mathtt{OUT}_i(\mathit{loop})$

Must eventually reach the fixpoint in a finite # of iterations

Should reach the desired fixpoint

Iterating won't work for arbitrary GEN and KILL functions
Will work for **monotonic** functions over **finite-height domains**

Reaching definitions example