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

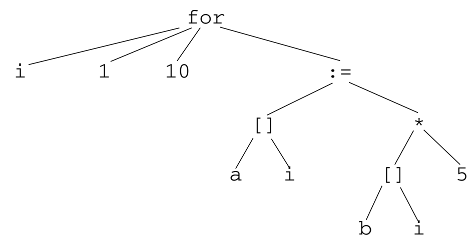
Examples

- (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)

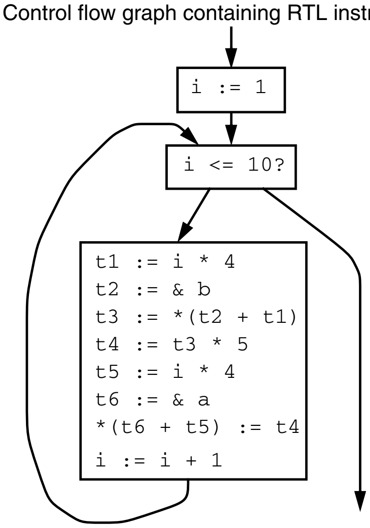
Standard RTL operators:

assignment	`x := y;`
unary op	`x :=` *op* `y;`
binary op	`x := y` *op* `z;`
address-of	`p := &y;`
load	`x := *(p + o);`
store	`*(p + o) := x;`
call	`x := f(...);`
unary compare	*op* `x ?`
binary compare	`x` *op* `y ?`

Source:

```
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
  - 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?

Components of representation

Operations

Dependences between operations

- **control** dependences: sequencing of operations
  - evaluation of then & else arms depends on result of test
  - side-effects of statements occur in right order
- **data** dependences: flow of values from **definitions** to **uses**
  - operands computed before operation

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**
  - 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 $\Rightarrow$ analysis can be slow

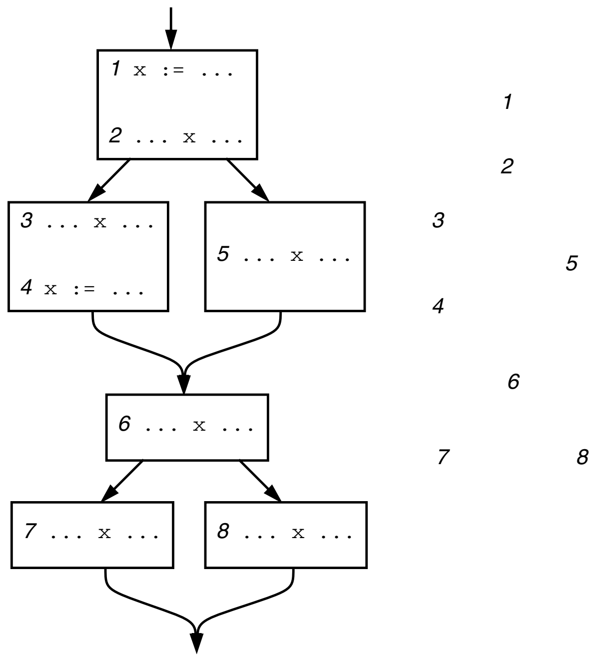
Option 2: def/use chains, linking each def to each use

+ explicit $\Rightarrow$ 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)
- ...

Example



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.:
$\{x \to s_1, y \to s_5, y \to s_8\}$

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 := ...`:
$$\text{info}_{succ} = \text{info}_{pred} - \{x \to s' \mid \forall s'\} \cup \{x \to s\}$$

A pointer assignment may modify anything, but doesn't definitely replace anything
- **weak update**

s: `*p := ...`:
$$\text{info}_{succ} = \text{info}_{pred} \cup \{x \to s \mid \forall x \in \text{may-point-to}(p)\}$$

Other statements: do nothing
$$\text{info}_{succ} = \text{info}_{pred}$$

Constraints for reaching definitions, continued

Branches pass through reaching defs to both successors

$info_{succ[i]} = info_{pred}$, $\forall i$

Merges take the union of all incoming reaching defs

- we don't know which path is being taken at run-time
  $\Rightarrow$ be conservative

$info_{succ} = \cup_i\ info_{pred[i]}$

Conditions at entry to CFG: definitions of formals

$info_{entry} = \{x \rightarrow \mathit{entry} \mid \forall x \in \text{formals}\}$

Solving constraints

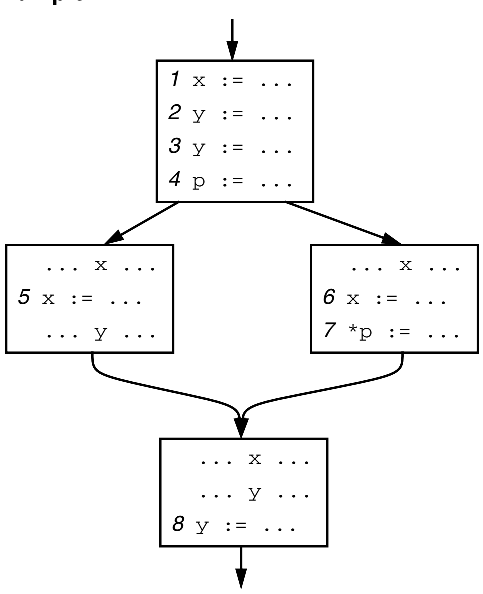
A given program yields a system of constraints

Need to solve constraints

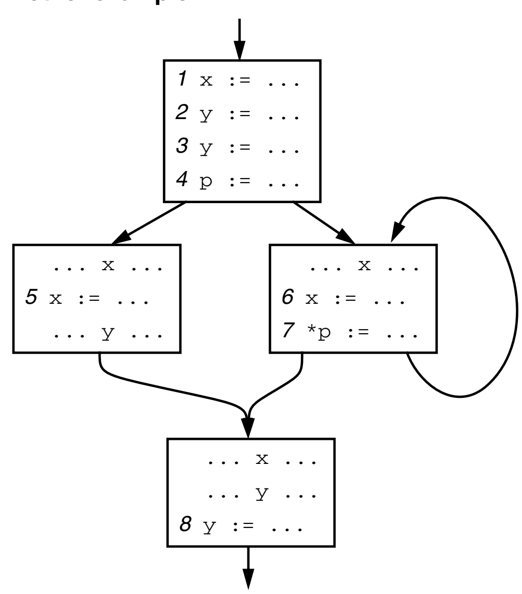
For reaching definitions,
can traverse instructions in forward topological order,
computing successor info from predecessor info

- because of how the constraints are defined

Example



Another example



Topological order not defined!

Loop terminology

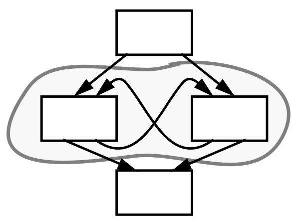
loop: strongly-connected component in CFG with single entry

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

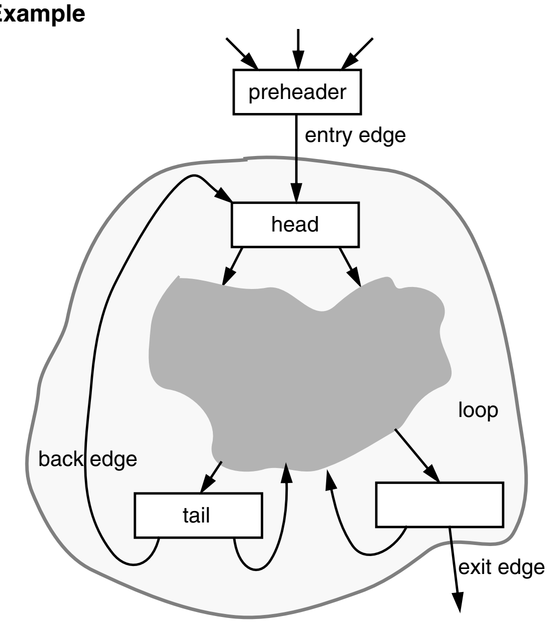
loop head node: target of loop entry edge
loop tail node: source of back edge
loop preheader node:
single node that's source of loop entry edge

nested loop: loop whose head is inside another loop

reducible flow graph: all SCC's have single entry



Example



Analysis of loops

If CFG has a loop, data flow constraints are recursively defined:
$\text{info}_{\text{loop-head}} = \text{info}_{\text{loop-entry}} \cup \text{info}_{\text{back-edge}}$
$\text{info}_{\text{back-edge}} = \ldots\ \text{info}_{\text{loop-head}} \ldots$

Substituting definition of $\text{info}_{\text{back-edge}}$:
$\text{info}_{\text{loop-head}} = \text{info}_{\text{loop-entry}} \cup (\ldots\ \text{info}_{\text{loop-head}} \ldots)$

Summarizing r.h.s. as F:
$\text{info}_{\text{loop-head}} = F(\text{info}_{\text{loop-head}})$

A legal solution to constraints is a **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:
   $\text{info}_{\text{loop-head}} = \textit{guess}$

2. Solve equations for loop body:
   $\text{info}_{\text{back-edge}} = F_{body}(\text{info}_{\text{loop-head}})$
   $\text{info}_{\text{loop-head}}' = \text{info}_{\text{loop-entry}} \cup \text{info}_{\text{back-edge}}$

3. Test if found fixed-point:
   $\text{info}_{\text{loop-head}}' = \text{info}_{\text{loop-head}}$?

   A. if same, then done

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

   $\text{info}_{\text{back-edge}}' = F_{body}(\text{info}_{\text{loop-head}}')$
   $\text{info}_{\text{loop-head}}'' = \text{info}_{\text{loop-entry}} \cup \text{info}_{\text{back-edge}}'$
   $\text{info}_{\text{loop-head}}'' = \text{info}_{\text{loop-head}}'$?
   ...

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: $info_{loop\text{-}head} = info_{loop\text{-}entry}$

(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