CS 412
Introduction to Compilers

Andrew Myers
Cornell University

Lecture 27: Dataflow analysis
6 Apr 01

Need for dataflow analysis

- Most optimizations require program analysis to determine safety
- This lecture: dataflow analysis
- Standard program analysis framework

Dataflow analyses

- **Live variable analysis** — register allocation, dead-code elimination
- **Reaching definitions**: what points in program does each variable definition reach? — copy, constant propagation
- **Available expressions**: which expressions computed earlier still have same value? — common sub-expression elimination

IR for data-flow analysis

- Tree IR: good for instruction selection, not so good for dataflow analysis
- Can flatten tree representation into simple nodes (a,b,c temps, labels L)

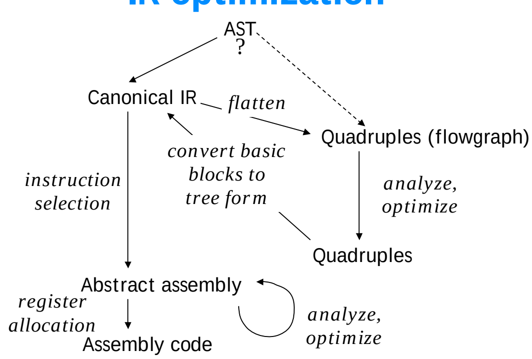
MOVE(a, OP(b,c))	a = b OP c
MOVE(a, MEM(b))	a = [b]
MOVE(MEM(a), b)	[a] = b
JUMP(L)	goto L
CJUMP(a,L1,L2)	if a goto L1 else L2
LABEL(L)	L:
MOVE(a, CALL(f,...))	a = f(...)
EXP(a, CALL(f,...))	f(...)

Quadruples

Quadruples

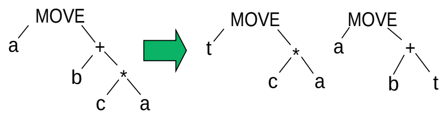
- Quadruple sequence is control flow graph (flowgraph)
- Nodes in graph: quadruples (not assembly statements)
- Edges in graph: ways to transfer control between quadruples (including fall-through)
- For node *n*, *use*[*n*] is variables used, *def* [*n*] is variables defined (assigned)
- Can generate directly from AST

IR optimization



Converting to quadruples

- Conversion is tree simplification that aggressively adds new temporaries



a = b + (c * a)

t = c * a
a = b + t

Converting back to tree

- Convert quadruples to simple trees
- Look for temporaries in statement sequence used and defined only once
- Move definition just before use
- Glue tree, eliminating temporary

t = c * a
...
a = b + t

⇒

MOVE(t, *(c,a))
...
MOVE(a, +(b,t))

⇒

MOVE(a, +(b,*(c,a)))

- Requires dataflow analyses to do right (reaching definitions, available expressions)

Def & Use

n	*def*[*n*]	*use*[*n*]
a = b OP c	a	b,c
a = [b]	a	b
[a] = b		a, b
goto L		
if a goto L1 else goto L2		a
L:		
a = f(...)	a	...
f(...)		...

Live variable analysis

- Useful even for IR: dead code elimination
- Output: $in[n]$ and $out[n]$ associated with every node n in flowgraph
- Constraints:

  $in[n] \supseteq use[n]$
  $in[n] \cup def[n] \supseteq out[n]$
  $out[n] \supseteq in[n']$ for all successors n' of n

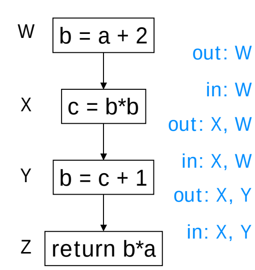
- Dataflow equations:

  $in[n] = use[n] \cup (out[n] - def[n])$
  $out[n] = \cup_{n'} \; in[n']$

Reaching definitions analysis

- Question: what uses in program does a given variable definition reach?
- Used for constant propagation, copy propagation
  - if only one definition reaches a particular use, can replace use by definition
  - copy propagation requires that copied value still has same value – use *available expressions*
- Input: flowgraph
- Output: $in[n]$, $out[n]$ is set of nodes defining some variable such that defn may reach beginning, end of n

Reaching definitions



Gen, kill

- Define: $defs(x)$ is set of nodes defining var x
- Define: $gen[n]$, $kill[n]$

n	$gen[n]$	$kill[n]$
a = b OP c	{ n }	$defs$(a) - { n }
a = [b]	{ n }	$defs$(a) - { n }
[a] = b	{ }	{ }
goto L	{ }	{ }
if a goto L1 else goto L2	{ }	{ }
L:	{ }	{ }
a = f(...)	{ n }	$defs$(a) - { n }
f(...)	{ }	{ }

Solution Constraints

$out[n] \supseteq gen[n]$

"A definition made by n at least reaches n's output"

$in[n'] \supseteq out[n]$ (if n' is successor of n)

"definitions reach node n' if they exit *any* predecessor n"

$out[n] \cup kill[n] = in[n]$

"A definition that reaches the input either reaches the output or is killed"

Data-flow equations

$$in[n'] = \bigcup_{n \in prev[n']} out[n]$$

$$out[n] = gen[n] \cup (in[n] - kill[n])$$

- Algorithm: init $in[n]$, $out[n]$ with empty sets, apply equations as assignments until no progress (usual representation: bit vector)
- Eventually all equations satisfied
- Will terminate because $in[n]$, $out[n]$ can only grow, can be no larger than set of all defns
- Finds minimal solution to constraint eqns: accurate

Def-use chains

- Reaching definitions tells which nodes a def can reach
- If node *uses* same variable, definition affects node (conservatively)
- Def-use (du-) chain: def node + all nodes with affected uses
- Use-def (ud-) chain: use node + all nodes with defs that might affect use

du-, ud-chains

DU

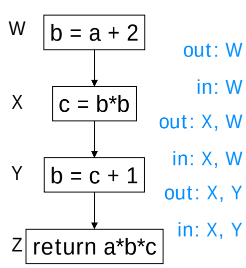
b: (W→X), (Y→Z)

c: (X→Y, Z)

UD

b: (X←W), (Z←Y)

c: (Y←X), (Z←X)



Webs

- du-chain, ud-chain *intersect* if share some use or definition
- *web* : maximal set of intersecting du, ud-chains
  - disjoint set union algorithm with path compression: computable in nearly linear time
- Same variable may comprise multiple non-interacting webs: permits more optimization

Webs

- Register allocation by webs avoids false conflicts

```
int i;
for (i = 0; i<n; i++) { ... }
...
for (i = 0; i<n; i++) { ... }
```

no use/def pairs!

- Two different webs: can allocate i to two different registers

Register allocation

1. use reaching definitions to compute all related uses and defs
2. compute disjoint webs, rename all temporaries to their web names
3. run register allocation as before : fewer interfering temporaries

Forward vs. Backward

- Liveness: backward analysis

$$in[n] = use[n] \cup (out[n] - def[n])$$
$$out[n] = \bigcup_{n' \in succ[n]} in[n']$$

- Reaching definitions: forward analysis

$$out[n] = gen[n] \cup (in[n] - kill[n])$$
$$in[n'] = \bigcup_{n \in prev[n']} out[n]$$

Dataflow analysis

- Many dataflow analyses characterized simply by
  - forward *vs.* backward analysis
  - *gen*[*n*]
  - *kill*[*n*]
  - Use of intersection vs. union when combining data from several nodes (operator $\sqcap$)

$$out[n] = gen[n] \cup (in[n] - kill[n])$$
$$in[n'] = \sqcap_{n \in prev[n']} out[n]$$

Available expressions

- Idea: want to perform common subexpression elimination

```
a = x+1            a = x+1
  ...      ⇨         ...
b = x+1            b = a
```

- Transformation is safe if original x+1 is an *available expression* (still computes same value)

Dataflow values

- Let *in*[*n*], *out*[*n*] be sets of nodes whose computed expression is available at *n*

n	*gen*[*n*]	*kill*[*n*]
a=b *OP* c	{*n*} – *kill*[*n*]	*uses*(a)
a=[b]	{*n*} – *kill*[*n*]	*uses*(a)
[a]=b	{}	*uses*([x]) (for all x that may be equal to a)
a=f(b_1,...b_n)	{}	*uses*([x]) (for all x)
other	{}	{}

Constraints

$out[n] \supseteq gen[n]$

"An expression made available by n at least reaches n's output"

$in[n'] \subseteq out[n]$ (if n' is succ. of n)

"An expression is available at n' only if it is available at *every* predecessor n"

$out[n] \cup kill[n] \supseteq in[n]$

"An expression available on input is either available on output or killed"

Dataflow equations

$out[n] \supseteq gen[n]$

$in[n'] \subseteq out[n]$ (if n' is succ. of n)

$out[n] \cup kill[n] \supseteq in[n]$

Equations for iterative solution:

$out[n] = gen[n] \cup (in[n] - kill[n])$

$in[n'] = \bigcap_{n \in \mathrm{pred}[n']} out[n]$

$\sqcap = \cap$ Starting condition:

$in[n]$ is set of *all* nodes

$in[start] = \varnothing$

Summary

- Tree IR makes dataflow more difficult
- Saw reaching definitions, available expressions analyses
- How to use reaching definitions for better register allocations via webs
- *Next time:* a theory to explain why iterative solving works