Introduction to Dataflow Analysis

15-745 Optimizing Compilers
Spring 2006

Peter Lee

On to global optimizations!

- Most of the important optimization opportunities (e.g., in loops) are not local
- We'll start with a simple global optimization: simple constant propagation
- Then develop a general framework — dataflow analysis — that supports this and other key optimizations

Classic reference: Matthew S. Hecht, "Flow Analysis of Computer Programs, Elsevier Science, NY, 1977

Simple constant propagation

```
a = 5;                          a = 5;
b = 3;                          b = 3;
 ...                             ...
n = a + b;                      n = 5 + 3;
for (i=0; i<n; ++i)             for (i=0; i<n; ++i)
  ..                              ..
```

If a and b can be determined to be constants, then replace them

Must analyze the code, to determine if it is legal to perform this program transformation

A sample program

```
int fib10 () {
  var n, old, older,
      result, i : int;

  n = 10;
  older = 0;
  old = 1;
  result = 0;

  if (n <= 1) return n;

  for (i=2; i<n; i+=1) {
    result = old + older;
    older = old;
    old = result;
  }

  return result;
}
```

source code

```
MOVE(TEMP(15), CONST(10))
MOVE(TEMP(13), CONST(0))
MOVE(TEMP(14), CONST(1))
MOVE(TEMP(12), CONST(0))
CJUMP(LE, TEMP(15), CONST(1), .L8, .L7)
LABEL(.L8)
MOVE(TEMP(1), TEMP(15))
JUMP(.L1)
LABEL(.L7)
MOVE(TEMP(11), CONST(2))
JUMP(.L3)
LABEL(.L3)
CJUMP(LT, TEMP(11), TEMP(15), .L4, .L5)
LABEL(.L4)
MOVE(TEMP(12), BINOP(PLUS, TEMP(14), TEMP(13)))
MOVE(TEMP(13), TEMP(14))
MOVE(TEMP(14), TEMP(12))
MOVE(TEMP(11), BINOP(PLUS, TEMP(11), CONST(1)))
JUMP(.L3)
LABEL(.L5)
MOVE(TEMP(1), TEMP(12))
JUMP(.L1)
```

linearized IR

A sample program

```
int fib10 () {
  var n, old, older,
      result, i : int;

  n = 10;
  older = 0;
  old = 1;
  result = 0;

  if (n <= 1) return n;

  for (i=2; i<n; i+=1) {
    result = old + older;
    older = old;
    old = result;
  }

  return result;
}
```

source code

```
n = 10
older = 0
old = 1
result = 0
if n > 1 then goto 7
return n
i = 2
if i > n goto 14
result = old + older
older = old
old = result
i = i + 1
goto 8
return result
```

easier-to-read version of linearized IR

A sample program

```
int fib10 () {
  var n, old, older,
      result, i : int;

  n = 10;
  older = 0;
  old = 1;
  result = 0;

  if (n <= 1) return n;

  for (i=2; i<n; i+=1) {
    result = old + older;
    older = old;
    old = result;
  }

  return result;
}
```

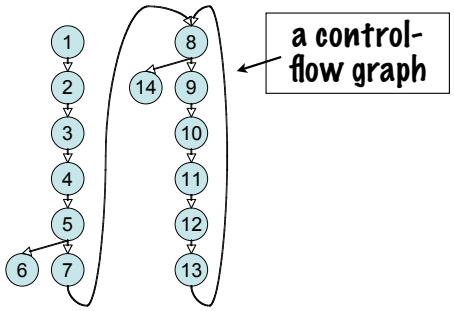
source code

```
1:  n = 10
2:  older = 0
3:  old = 1
4:  result = 0
5:  if n > 1 then goto 7
6:  return n
7:  i = 2
8:  if i > n goto 14
9:  result = old + older
10: older = old
11: old = result
12: i = i + 1
13: goto 8
14: return result
```

linearized IR, with program point labels

Terminology alert!

```
1:  n = 10
2:  older = 0
3:  old = 1
4:  result = 0
5:  if n > 1 then goto 7
6:  return n
7:  i = 2
8:  if i > n goto 14
9:  result = old + older
10: older = old
11: old = result
12: i = i + 1
13: goto 8
14: return result
```



a control-flow graph

E.g.: pred(8) = {7, 13}

pred(n): immediate predecessors of node n
succ(n): immediate successors of node n

More terminology

- A definition is a statement that defines a value for some variable v
  - e.g., d: v = x ⊕ y
- defs(v) = the set of definitions that define v

Yet more terminology...

* For each statement, in isolation, it is easy to see if it is definition
* Consider a statement at point d
  * d: v = x op y
* This statement generates a definition...
* ...and kills all other definitions that define a value for v

GENs and KILLs

```
                                  GEN?    KILLs
1:  n = 10                         Y      {}
2:  older = 0                      Y      {10}
3:  old = 1                        Y      {11}
4:  result = 0                     Y      {9}
5:  if n > 1 then goto 7           N      {}
6:  return n                       N      {}
7:  i = 2                          Y      {12}
8:  if i > n goto 14               N      {}
9:  result = old + older           Y      {4}
10: older = old                    Y      {2}
11: old = result                   Y      {3}
12: i = i + 1                      Y      {7}
13: goto 8                         N      {}
14: return result                  N      {}
```

for a statement d that defines v,
KILL(d) = defs(v) - {d}

Reaching definitions

A definition at program point d reaches program point u if there is a control-flow path from d to u that does not contain a definition of the same variable as d

```
1:  n = 10
2:  older = 0
3:  old = 1
4:  result = 0
5:  if n > 1 then goto 7
6:  return n
7:  i = 2
8:  if i > n goto 14
9:  result = old + older
10: older = old
11: old = result
12: i = i + 1
13: goto 8
14: return result
```

Does 1 reach 5?
What uses of n are reached by 1?
What definitions reach 8?

Reaching definitions, formally

$$rd(n) = \bigcup_{p \in pred(n)} (gen(p) \cup (rd(p) - kill(p)))$$

* rd(n) = the set of definitions that reach statement n

Alternatively: In and Out sets

$$in(n) = \bigcup_{p \in pred(n)} out(p)$$

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

- in(n): the set of defs that reach the beginning of node n
- out(n): the set of defs that reach the end of node n

Solving reaching definitions

- In general, we won't be able to compute an exact solution to reaching definitions
- We want, therefore, a conservative approximation
  - if d reaches n for some execution of P, then d∈rd(n)
  - so, rd(n) = {1, 2, ..., 14} would work for our current example (though obviously we want to do better)

```
...
if (...)
  x = 1;
...
a = x
...
```

Fixed point solutions

$$in(n) = \bigcup_{p \in pred(n)} out(p)$$

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

- Notice, informally:
  - in() and out() are monotonic (increasing)
  - finite number of definitions
- So, can initialize in()=out()={}, for all n, and the find a fixed point iteratively

Warning!

- We are being dangerously informal here
  - what is meant by "conservative approximation"
  - why is a fixed point solution to rd(n) a reasonable solution, and is it really conservative?
- We will definitely need to address these and other related questions (next time)

$$in(n) = \bigcup_{p \in pred(n)} out(p)$$

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

		GEN?	KILLs	IN0	OUT0
0:	entry	N	{}	{}	{}
1:	n = 10	Y	{}	{}	{}
2:	older = 0	Y	{10}	{}	{}
3:	old = 1	Y	{11}	{}	{}
4:	result = 0	Y	{9}	{}	{}
5:	if n > 1 then goto 7	N	{}	{}	{}
6:	return n	N	{}	{}	{}
7:	i = 2	Y	{12}	{}	{}
8:	if i > n goto 14	N	{}	{}	{}
9:	result = old + older	Y	{4}	{}	{}
10:	older = old	Y	{2}	{}	{}
11:	old = result	Y	{3}	{}	{}
12:	i = i + 1	Y	{7}	{}	{}
13:	goto 8	N	{}	{}	{}
14:	return result	N	{}	{}	{}

$$in(n) = \bigcup_{p \in pred(n)} out(p)$$

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

		GEN?	KILLs	IN1	OUT1
0:	entry	N	{}	{}	{}
1:	n = 10	Y	{}	{}	{1}
2:	older = 0	Y	{10}	{1}	{1,2}
3:	old = 1	Y	{11}	{1,2}	{1,2,3}
4:	result = 0	Y	{9}	{1,2,3}	{1,2,3,4}
5:	if n > 1 then goto 7	N	{}	{1,2,3,4}	{1,2,3,4}
6:	return n	N	{}	{1,2,3,4}	{1,2,3,4}
7:	i = 2	Y	{12}	{1,2,3,4}	{1,2,3,4,7}
8:	if i > n goto 14	N	{}	{1,2,3,4,7}	{1,2,3,4,7}
9:	result = old + older	Y	{4}	{1,2,3,4,7}	{1,2,3,7,9}
10:	older = old	Y	{2}	{1,2,3,7,9}	{1,3,7,9,10}
11:	old = result	Y	{3}	{1,3,7,9,10}	{1,7,9,10,11}
12:	i = i + 1	Y	{7}	{1,7,9,10,11}	{1,9,10,11,12}
13:	goto 8	N	{}	{1,9,10,11,12}	{1,9,10,11,12}
14:	return result	N	{}	{1,2,3,4,7}	{1,2,3,4,7}

$$in(n) = \bigcup_{p \in pred(n)} out(p)$$

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

		GEN?	KILLs	IN2	OUT2
0:	entry	N	{}	{}	{}
1:	n = 10	Y	{}	{}	{1}
2:	older = 0	Y	{10}	{1}	{1,2}
3:	old = 1	Y	{11}	{1,2}	{1,2,3}
4:	result = 0	Y	{9}	{1,2,3}	{1,2,3,4}
5:	if n > 1 then goto 7	N	{}	{1,2,3,4}	{1,2,3,4}
6:	return n	N	{}	{1,2,3,4}	{1,2,3,4}
7:	i = 2	Y	{12}	{1,2,3,4}	{1,2,3,4,7}
8:	if i > n goto 14	N	{}	{1-4,7,9-12}	{1-4,7,9-12}
9:	result = old + older	Y	{4}	{1-4,7,9-12}	{1-3,7,9-12}
10:	older = old	Y	{2}	{1-3,7,9-12}	{1,3,7,9-12}
11:	old = result	Y	{3}	{1,3,7,9-12}	{1,7,9-12}
12:	i = i + 1	Y	{7}	{1,7,9-12}	{1,9-12}
13:	goto 8	N	{}	{1,9-12}	{1,9-12}
14:	return result	N	{}	{1-4,7,9-12}	{1-4,7,9-12}

$$in(n) = \bigcup_{p \in pred(n)} out(p)$$

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

		GEN?	KILLs	IN3	OUT3
0:	entry	N	{}	{}	{}
1:	n = 10	Y	{}	{}	{1}
2:	older = 0	Y	{10}	{1}	{1,2}
3:	old = 1	Y	{11}	{1,2}	{1,2,3}
4:	result = 0	Y	{9}	{1,2,3}	{1,2,3,4}
5:	if n > 1 then goto 7	N	{}	{1,2,3,4}	{1,2,3,4}
6:	return n	N	{}	{1,2,3,4}	{1,2,3,4}
7:	i = 2	Y	{12}	{1,2,3,4}	{1,2,3,4,7}
8:	if i > n goto 14	N	{}	{1-4,7,9-12}	{1-4,7,9-12}
9:	result = old + older	Y	{4}	{1-4,7,9-12}	{1-3,7,9-12}
10:	older = old	Y	{2}	{1-3,7,9-12}	{1,3,7,9-12}
11:	old = result	Y	{3}	{1,3,7,9-12}	{1,7,9-12}
12:	i = i + 1	Y	{7}	{1,7,9-12}	{1,9-12}
13:	goto 8	N	{}	{1,9-12}	{1,9-12}
14:	return result	N	{}	{1-4,7,9-12}	{1-4,7,9-12}

done!

Worklist algorithm

```
initialize: in(n) = out(b) = {}, all b
initialize: in(entry) = {}

workqueue W = all blocks - {entry}

while (W not empty) do
    remove b from W
    old = out(b)
    in(b) = ⋃ out(p)
            p∈pred(b)
    out(b) = gen(b) U (in(b) - kill(b))
    if (old != out(b)) then
        W = W U succ(b)
```

Lots and lots of questions

- Theoretical
  - correctness of this method?
  - termination?
  - efficiency?
  - other analysis problems?
- We'll address these theoretical questions next time
- Today, we look at a few practical matters

Practical matters

- What order to visit the nodes?
- Iteration over basic blocks, not statements?
- Efficient representation of sets?
- Representing reaching definitions info?

Node visit order

- In theory, it doesn't matter what order the nodes are visited
- But as a practical matter, visit order affects how quickly the iterative analysis converges
- Reaching definitions is a forward dataflow problem; they tend to converge most quickly when nodes are visited in "forward" order

$in(n) = \bigcup_{p \in pred(n)} out(p)$

Visiting in reverse order, 14...0

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

		GEN?	KILLs	IN0	OUT0
0:	entry	N	{}	{}	{}
1:	n = 10	Y	{}	{}	{}
2:	older = 0	Y	{10}	{}	{}
3:	old = 1	Y	{11}	{}	{}
4:	result = 0	Y	{9}	{}	{}
5:	if n > 1 then goto 7	N	{}	{}	{}
6:	return n	N	{}	{}	{}
7:	i = 2	Y	{12}	{}	{}
8:	if i > n goto 14	N	{}	{}	{}
9:	result = old + older	Y	{4}	{}	{}
10:	older = old	Y	{2}	{}	{}
11:	old = result	Y	{3}	{}	{}
12:	i = i + 1	Y	{7}	{}	{}
13:	goto 8	N	{}	{}	{}
14:	return result	N	{}	{}	{}

$in(n) = \bigcup_{p \in pred(n)} out(p)$

Visiting in reverse order, 14...0

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

		GEN?	KILLs	IN1	OUT1
0:	entry	N	{}	{}	{}
1:	n = 10	Y	{}	{}	{1}
2:	older = 0	Y	{10}	{}	{2}
3:	old = 1	Y	{11}	{}	{3}
4:	result = 0	Y	{9}	{}	{4}
5:	if n > 1 then goto 7	N	{}	{}	{}
6:	return n	N	{}	{}	{}
7:	i = 2	Y	{12}	{}	{7}
8:	if i > n goto 14	N	{}	{}	{}
9:	result = old + older	Y	{4}	{}	{9}
10:	older = old	Y	{2}	{}	{10}
11:	old = result	Y	{3}	{}	{11}
12:	i = i + 1	Y	{7}	{}	{12}
13:	goto 8	N	{}	{}	{}
14:	return result	N	{}	{}	{}

$in(n) = \bigcup_{p \in pred(n)} out(p)$

Visiting in reverse order, 14...0

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

		GEN?	KILLs	IN2	OUT2
0:	entry	N	{}	{}	{}
1:	n = 10	Y	{}	{}	{1}
2:	older = 0	Y	{10}	{1}	{1,2}
3:	old = 1	Y	{11}	{2}	{2,3}
4:	result = 0	Y	{9}	{3}	{3,4}
5:	if n > 1 then goto 7	N	{}	{4}	{4}
6:	return n	N	{}	{}	{}
7:	i = 2	Y	{12}	{}	{7}
8:	if i > n goto 14	N	{}	{7}	{7,12}
9:	result = old + older	Y	{4}	{}	{9}
10:	older = old	Y	{2}	{9}	{9,10}
11:	old = result	Y	{3}	{10}	{10,11}
12:	i = i + 1	Y	{7}	{11}	{11,12}
13:	goto 8	N	{}	{12}	{12}
14:	return result	N	{}	{}	{}

$in(n) = \bigcup_{p \in pred(n)} out(p)$

Visiting in reverse order, 14...0

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

still not done!

		GEN?	KILLs	IN3	OUT3
0:	entry	N	{}	{}	{}
1:	n = 10	Y	{}	{}	{1}
2:	older = 0	Y	{10}	{1}	{1,2}
3:	old = 1	Y	{11}	{2}	{1,2,3}
4:	result = 0	Y	{9}	{3}	{2,3,4}
5:	if n > 1 then goto 7	N	{}	{3,4}	{3,4}
6:	return n	N	{}	{4}	{4}
7:	i = 2	Y	{12}	{4}	{4,7}
8:	if i > n goto 14	N	{}	{7,11,12}	{7,11,12}
9:	result = old + older	Y	{4}	{7,12}	{7,9,12}
10:	older = old	Y	{2}	{9}	{9,10,12}
11:	old = result	Y	{3}	{9,10}	{9,10,11}
12:	i = i + 1	Y	{7}	{10,11}	{10,11,12}
13:	goto 8	N	{}	{11,12}	{11,12}
14:	return result	N	{}	{7,12}	{7,12}

Practical matters

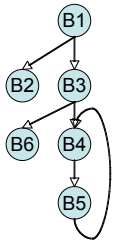
- ✓ What order to visit the nodes?
- Iteration over basic blocks, not statements?
- Efficient representation of sets?
- Representing reaching definitions info?

Basic block nodes

- For straight-line code:
  - in[s1; s2] = in[s1]
  - out[s1; s2] = out[s2]
- So, for each basic block we can compute (in a linear pass) the Gen and Kill sets, and then perform the iterative analysis on the blocks instead of individual statements
- Finally perform a linear analysis for each statement in each block

Basic blocks

1: n = 10 2: older = 0 3: old = 1 4: result = 0 5: if n > 1 then goto 7	B1
6: return n	B2
7: i = 2	B3
8: if i > n goto 14	B4
9: result = old + older 10: older = old 11: old = result 12: i = i + 1 13: goto 8	B5
14: return result	B6



Basic blocks, cont'd

		GEN	KILL
1: n = 10 2: older = 0 3: old = 1 4: result = 0 5: if n > 1 then goto 7	B1	{1,2,3,4}	{9,10,11}
6: return n	B2	{}	{}
7: i = 2	B3	{7}	{12}
8: if i > n goto 14	B4	{}	{}
9: result = old + older 10: older = old 11: old = result 12: i = i + 1 13: goto 8	B5	{9,10,11,12}	{2,3,4,7}
14: return result	B6	{}	{}

Typically, far fewer nodes in the control-flow graph, hence much faster convergence in practice

Practical matters

- ✓ What order to visit the nodes?
- ✓ Iteration over basic blocks, not statements?
- Efficient representation of sets?
- Representing reaching definitions info?

Bit vectors

- Bit vectors are commonly used to represent the gen, kill, in, and out sets
  - each definition is a bit position
  - gen: 1 in each position generated, else 0
  - kill: 0 in each position killed, else 1
  - out(n) = in(n) $\vee$ (gen(n) $\wedge$ kill(n))

Practical matters

- ✓ What order to visit the nodes?
- ✓ Iteration over basic blocks, not statements?
- ✓ Efficient representation of sets?
- Representing reaching definitions info?

Use-def chains

```
1:  n = 10
2:  older = 0
3:  old = 1
4:  result = 0
5:  if i > 1 then goto 7
6:  return n
7:  i = 2
8:  if i > n goto 14
9:  result = old + older
10: older = old
11: old = result
12: i = i + 1
13: goto 8
14: return result
```

For each use of var v in a statement s, a list of definitions of v that reach s

Note that def-use chains are also useful, but NOT given by reaching definitions analysis

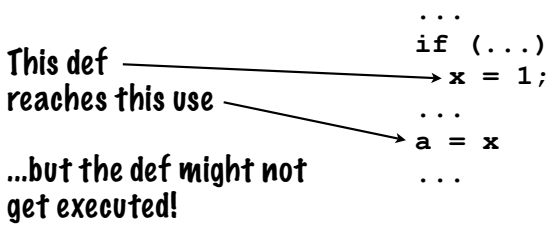
Practical matters

- ✓ What order to visit the nodes?
- ✓ Iteration over basic blocks, not statements?
- ✓ Efficient representation of sets?
- ✓ Representing reaching definitions info?

Simple constant propagation

- Calculate reaching definitions
- For a use of v in statement n, if the only definition of v that reaches n is of the form
  - v = c
- for a constant c, then replace v with c

Uninitialized variables?



Languages like C typically do not define the behavior of programs with uninitialized variables, so simple constant propagation is OK

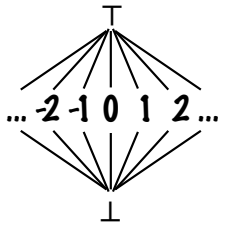
Q: How would you use RD to detect possible uninitialized variables?

Better constant propagation

- In general, however, we will want to deal with uninitialized variables in a more principled way
- In general, this is done by defining a lattice and then binding each dataflow value with a lattice element

Constant propagation lattice

- All variables initialized to ⊥
- If defs d1: v=c1 and d2: v=c2 reach n, then v is join(c1,c2)
- For gen d: v = c, bind d with c
- For gen d: v = x⊕y, bind d with T
  - or calculate the lattice values of x and y and then join them



Next time

- Dead-code elimination
  - based on liveness analysis
- Common-subexpression elimination
  - based on available-expressions analysis
- The general dataflow analysis framework