

Classical Dataflow Optimizations

15-745 Optimizing Compilers
Spring 2006

Peter Lee

Reminders

- * Task 0 is due!
- * Task 1 is available!
 - * due in 2 weeks
 - * watch the RSS feed for bug reports and an advice column
- * Read up on dataflow analysis (8.1-8.4)

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**

Reaching definitions

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

- * $\text{gen}(n)$ = set of definitions generated by n
- * $\text{kill}(n)$ = set of definitions killed by n
 - * if n defines v , $\text{kill}(n) = \text{defs}(v) - \{n\}$
- * $\text{rd}(n)$ = the set of definitions that reach statement n

More common presentation

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

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

Using reaching definitions

- * For a use of variable v in statement n ,
 - * $n: x = \dots v \dots$
 - * if the definitions of v that reach n are all of the form
 - * $d: v = c$ [c a constant]
 - * then replace the use of v in n with c
 - * This is **simple (naïve) constant propagation**

Uninitialized variables?

This def reaches this use

...but the def might not get executed!

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

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

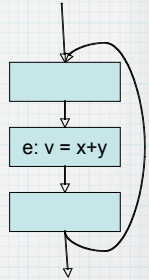
Q: How would you use RD to detect possible uninitialized variables?

Better constant propagation

- * In general, however, we will want to deal with uninitialized variables in a more principled way
- * This can be done by defining a lattice of dataflow values
- * We'll come back to this issue at the end of this lecture

Other applications

- * There are several important applications of reaching definitions analysis
- * Consider a loop containing an expr e
 - * if all of the reaching definitions of the operands of e are outside of the loop, then e can be moved out of the loop
- * This is **loop-invariant code motion**



Loop optimization

- * Loop-invariant code motion is just one example of a loop optimization
- * We will devote several future lectures to code motion and loop-optimization techniques

Live-out Variables

Liveness analysis

- * A variable **v** is **live-out** of statement **n** if **v** is used along some control path starting at **n**
- * otherwise, we say that **v** is **dead**

Liveness analysis

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

$$\text{out}(n) = \bigcup_{s \in \text{succ}(n)} \text{in}(s)$$

- * $\text{gen}(n)$ = the variables used by n
- * $\text{kill}(n)$ = if v is defined without using v , it is in $\text{kill}(n)$
- * $\text{in}(n)$ = set of variables live-in to n
- * $\text{out}(n)$ = set of variables live-out of n

Forward vs backward

- * Liveness analysis is a **backward analysis**, in contrast to reaching definitions, which uses a **forward analysis**

ud-chains and du-chains

- * Recall that we can build **ud-chains** from a reaching definitions analysis
- * link each use of a variable to the definitions that reach it
- * From a liveness analysis, it is straightforward to build **du-chains**
- * link each definition to its uses

du-chain example

```
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
```

Using liveness analysis

- * Suppose we have a statement
 - * $n: v = x \oplus y$
 - * where v is dead and \oplus has no side effects
- * Then n can be eliminated
- * This is **dead-code elimination**

Reminder: basic blocks

- * Remember that it is much more efficient to analyze basic blocks
 - * each basic block can be analyzed in a single linear scan to compute gen and kill
 - * then, perform the iterative analysis on the basic blocks
 - * finally, propagate each block's in and out sets to each statement

Available Expressions

Available-expressions analysis

- * Consider an expression, $x \oplus y$, occurring at one or more program points
- * Expression $x \oplus y$ is **available** at statement n if it is computed along every path from the entry node to n , and
- * on each path neither x nor y are modified after the last evaluation of $x \oplus y$

Available-expressions gen/kill

- * Let
 - * $\text{exprs}(v)$ = set of expressions that use v
- * Then
 - * $\text{gen}(n) = e$, if
 - * n evaluates e and,
 - * $e \notin \text{exprs}(v)$, if n defines v
 - * $\text{kill}(n) = \text{exprs}(v)$, if n defines v

Flow functions

- * In a dataflow analysis, we think of each node as defining a **flow function** (aka **transfer function**) that maps an **in set** to an **out set**
- * Exactly what flow function is defined by a node is determined by the semantics of the IR and the dataflow analysis problem
- * As a convenience, most flow functions are described in terms of in/out equations using $\text{gen}()$ and $\text{kill}()$

Gen and Kill for AE

Expr	Gen	Kill
$v = x + y$	$\{v = x + y\} - \text{exprs}(v)$	$\text{exprs}(v)$
$v = \text{mem}(a)$	$\{v = \text{mem}(a)\} - \text{exprs}(v)$	$\text{exprs}(v)$
$\text{mem}(a) = b$	$\{\}$	$\text{mem}(*)$
$v = f(a, \dots)$	$\{\}$	$\text{exprs}(v) + \text{mem}(*)$

Note: $\text{mem}(*)$ denotes all memory cells

We must take care with expressions that dereference pointers!

Without a pointer or alias analysis, usually we must be extremely conservative.

Available-expressions analysis

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

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

Note: a forward analysis.

out() is essentially the flow function.

in() defines what happens when two or more control path meet

AE example

0:	entry	in={}, out={}
B1	1: c = a+b 2: d = a*c 3: e = d*d 4: i = 1	in={}, out={a+b,a*c,d*d}
B2	5: f[i] = a+b 6: c = c*2 7: if c>d goto 10	in={a+b,d*d}, out={a+b,d*d,c>d}
B3	8: g[i] = d*d 9: goto 11	in={a+b,d*d,c>d}, out={a+b,d*d,c>d}
B4	10: g[i] = a*c 11: i = i+1	in={a+b,d*d,c>d}, out={a+b,d*d,c>d,a*c}
B5	12: if i<=10 goto 5 13: exit	in={a+b,d*d,c>d}, out={a+b,d*d,c>d,i*10}

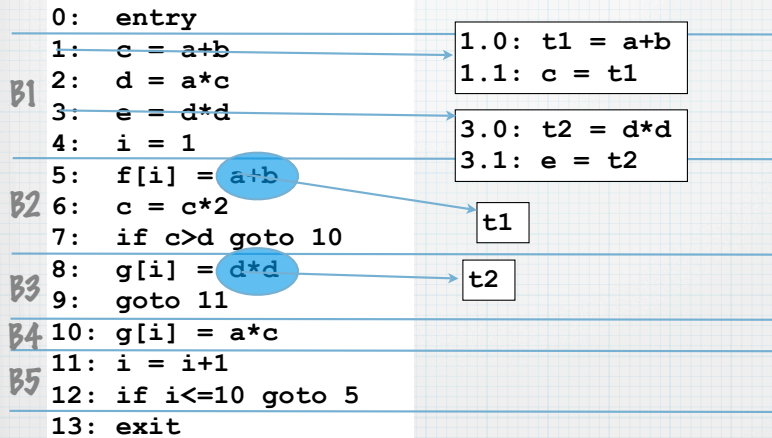
Using available expressions

- * Consider a statement, $n:v=x+y$, where $x+y$ is available
- * for every statement, $e:t=x+y$, whose expression reaches n , we can
 - * rewrite e as $e1:t'=x+y$; $e2:t=t'$
 - * and then replace $n:v=x+y$ with $n:v=t'$
- * This is **common-subexpression elimination**

AE example

0:	entry	in={}, out={}
B1	1: c = a+b 2: d = a*c 3: e = d*d 4: i = 1	in={}, out={a+b,a*c,d*d}
B2	5: f[i] = a+b 6: c = c*2 7: if c>d goto 10	in={a+b,d*d}, out={a+b,d*d,c>d}
B3	8: g[i] = d*d 9: goto 11	in={a+b,d*d,c>d}, out={a+b,d*d,c>d}
B4	10: g[i] = a*c 11: i = i+1	in={a+b,d*d,c>d}, out={a+b,d*d,c>d,a*c}
B5	12: if i<=10 goto 5 13: exit	in={a+b,d*d,c>d}, out={a+b,d*d,c>d,i*10}

AE example



Reaching expressions

- * The reaching expressions can be computed via a dataflow analysis
- * Typically, however, it is sufficient to do a simple backward traversal of the control-flow graph, searching for the available expressions or the definitions that kill

Our analyses so far

	union	intersection
forward	reaching definitions	available expressions
backward	live variables	

Back to Constant Propagation

Ingredients of a dataflow analysis

- * forward or backward?
- * flow/transfer function
- * meet operator
- * dataflow values

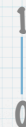
We have been waving our hands about this one...

Dataflow values

- * To simplify matters, consider programs with just one value
- * e.g., for reaching definitions, a program containing just one definition
- * e.g., for liveness analysis, a program with just one variable
- * e.g., for available expressions, a program with just one expression
- * In these cases, the dataflow values are $\{0,1\}$

Two-point lattice

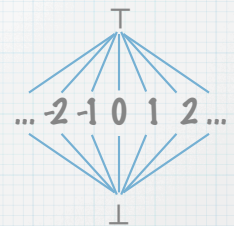
- * In fact, we think of this in terms of a two-point lattice
- * The join operator on this lattice is the “meet” operation of the specific dataflow analysis problem
- * RD: union/or
- * LA: union/or
- * AE: intersection/and



1
0

Constant-propagation lattice

- * For constant propagation, the dataflow values are more complicated
- * all constants
- * uninitialized variables
- * and so a more complex lattice is used



Foundations

- * In fact, the properties of the lattices used for a dataflow analysis are critical in determining soundness and termination
- * We will develop this theory further next time...