

More Dataflow Analysis

Monday, November 14, 2011

Recall steps to building analysis

- Step 1: Choose lattice
- Step 2: Choose direction of dataflow (forward or backward)
- Step 3: Create monotonic transfer function
- Step 4: Choose *confluence* operator (i.e., what to do at merges)
 - Either join or meet in the lattice
- Let's walk through these steps for a new analysis

Monday, November 14, 2011

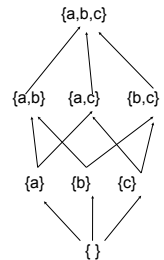
Liveness analysis

- Which variables are live at a particular program point?
- Used all over the place in compilers
 - Register allocation
 - Loop optimizations

Monday, November 14, 2011

Choose lattice

- What do we want to know?
- At each program point, want to maintain the set of variables that are live
- Lattice elements: sets of variables
- Natural choice for lattice: powerset of variables!



Monday, November 14, 2011

Choose dataflow direction

- A variable is *live* if it is used later in the program without being redefined
- At a given program point, we want to know information about what happens later in the program
- This means that liveness is a *backwards* analysis
 - Recall that we did liveness backwards when we looked at single basic blocks

Monday, November 14, 2011

Create x-fer functions

- What do we do for a statement like:
 $x = y + z$
- If x was live “before” (i.e., live after the statement), it isn't now (i.e., is not live before the statement)
- If y and z were not live “before,” they are now
- What about:
 $x = x$

Monday, November 14, 2011

Create x-fer functions

- Let's generalize
- For any statement s , we can look at which live variables are *killed*, and which new variables are made live (*generated*)
- Which variables are killed in s ?
 - The variables that are *defined* in s : $DEF(s)$
- Which variables are made live in s ?
 - The variables that are *used* in s : $USE(s)$
- If the set of variables that are live after s is X , what is the set of variables live before s ?

$$T_s(X) = use(s) \cup (X - def(s))$$

- Is this monotonic?

Monday, November 14, 2011

Dealing with aliases

- Aliases, as usual, cause problems
- Consider

```
int x, y
int *z, *w;
if (...) z = &y else z = &x
if (...) w = &y else w = &x
*z = *w; //which variable is defined? which is used?
```

- What should $USE(*z = *w)$ and $DEF(*z = *w)$ be?
 - Keep in mind: the goal is to get a list of variables that *may* be live at a program point
- For now, assume there is no aliasing

Monday, November 14, 2011

Dealing with function calls

- Similar problem as aliases:

```
int foo(int &x, int &y); //pass by reference!
```

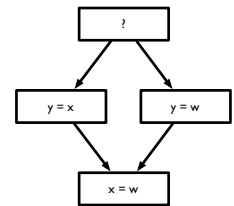
```
void main() {
  int x, y, z;
  z = foo(x, y);
}
```

- Simple solution: functions can do *anything* – redefine variables, use variables
 - So $DEF(foo())$ is $\{ \}$ and $USE(foo())$ is V
- Real solution: *interprocedural* analysis, which determines what variables are used and defined in foo

Monday, November 14, 2011

Choose confluence operator

- What happens at a merge point?
 - The variables live in to a merge point are the variables that are live along *either* branch
 - Confluence operator: Set union (\sqcup) of all live sets of outgoing edges



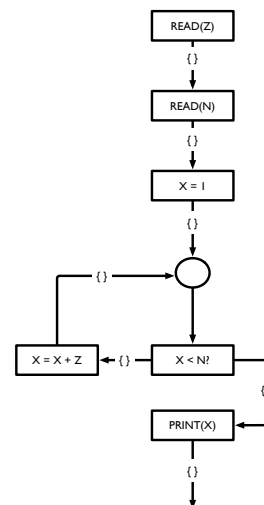
$$T_{merge} = \bigcup_{X \in succ(merge)} X$$

Monday, November 14, 2011

How to initialize analysis?

- At the end of the program, we know no variables are live → value at exit point is $\{ \}$
- What about elsewhere in the program?
 - We should initialize other sets to $\{ \}$
 - This is consistent with our approach to finding the least fixpoint

Monday, November 14, 2011



Monday, November 14, 2011

An alternate approach

- Dataflow analyses like live-variable analysis are *bit-vector* analyses: are even more structured than regular dataflow analysis
 - Consistent lattice: powerset
 - Consistent transfer functions
- Many sources only talk about bitvector dataflow

Monday, November 14, 2011

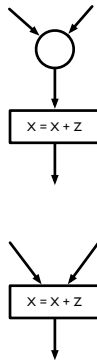
Bit-vector lattices

- Consider a single element, V , of the powerset(S) lattice
- Each item in S either appears in V or does not: can represent using a single bit
 - Can represent V as a *bit vector*
 - $\{a, b, c\} = \langle 1, 1, 1 \rangle$
 - $\{\} = \langle 0, 0, 0 \rangle$
 - $\{b, c\} = \langle 0, 1, 1 \rangle$
- \sqcup and \sqcap (which are just \cup and \cap) are simply bitwise \vee and \wedge , respectively

Monday, November 14, 2011

Eliminating merge nodes

- Many dataflow presentations do not use explicit merge nodes in CFG
- How do we handle this?
- Problem: now a node may be a statement *and* a merge point
- Solution: compose confluence operator and transfer functions
- Note: non-merge nodes have just one successor; this equation works for all nodes!



$$T(s) = \mathbf{use}(s) \cup \left(\bigcup_{X \in \text{succ}(s)} X \right) - \mathbf{def}(s)$$

Monday, November 14, 2011

Simplifying matters

$$T(s) = \mathbf{use}(s) \cup \left(\bigcup_{X \in \text{succ}(s)} X \right) - \mathbf{def}(s)$$

- Lets split this up into two different sets
 - $\text{OUT}(s)$: the set of variables that are live *immediately after* a statement is executed
 - $\text{IN}(s)$: the set of variables that are live *immediately before* a statement is executed

$$\begin{aligned} \text{IN}(s) &= \mathbf{use}(s) \cup (\text{OUT}(s) - \mathbf{def}(s)) \\ \text{OUT}(s) &= \bigcup_{t \in \text{succ}(s)} \text{IN}(t) \end{aligned}$$

Monday, November 14, 2011

Generalizing

- $\text{USE}(s)$ are the variables that become live due to a statement—they are *generated* by this statement
- $\text{DEF}(s)$ are the variables that stop being live due to a statement—they are *killed* by this statement

$$\begin{aligned} \text{IN}(s) &= \mathbf{gen}(s) \cup (\text{OUT}(s) - \mathbf{kill}(s)) \\ \text{OUT}(s) &= \bigcup_{t \in \text{succ}(s)} \text{IN}(t) \end{aligned}$$

Monday, November 14, 2011

Bit-vector analyses

- A bit-vector analysis is any analysis that
 - Operates over the powerset lattice, ordered by \subseteq and with \cup and \cap as its meet and join
 - Has transfer functions that can be written in the form:

$$\begin{aligned} \text{IN}(s) &= \mathbf{gen}(s) \cup (\text{OUT}(s) - \mathbf{kill}(s)) \\ \text{OUT}(s) &= \bigcup_{t \in \text{succ}(s)} \text{IN}(t) \end{aligned}$$

- Are these transfer functions monotonic? (Hint: if f and g are monotonic, is $f \cdot g$ monotonic?)
- \mathbf{gen} and \mathbf{kill} are dependent on the statement, but not on IN or OUT
- Things are a little different for forward analyses, and some analyses use \cap instead of \cup

Monday, November 14, 2011

Reaching definitions

- What definitions of a variable *reach* a particular program point
- A definition of variable x from statement s reaches a statement t if there is a path from s to t where x is not redefined
- Especially important if x is used in t
- Used to build *def-use* chains and *use-def* chains, which are key building blocks of other analyses
- Used to determine dependences: if x is defined in s and that definition reaches t then there is a flow dependence from s to t
- We used this to determine if statements were loop invariant
- All definitions that reach an expression must originate from outside the loop, or themselves be invariant

Monday, November 14, 2011

Creating a reaching-def analysis

- Can we use a powerset lattice?
- At each program point, we want to know which definitions have reached a particular point
- Can use powerset of set of definitions in the program
 - V is set of variables, S is set of program statements
 - Definition: $d \in V \times S$
 - Use a tuple, $\langle v, s \rangle$
- How big is this set?
 - At most $|V \times S|$ definitions

Monday, November 14, 2011

Forward or backward?

- What do you think?

Monday, November 14, 2011

Choose confluence operator

- Remember: we want to know if a definition *may* reach a program point
- What happens if we are at a merge point and a definition reaches from one branch but not the other?
 - We don't know which branch is taken!
 - We should union the two sets – any of those definitions can reach
- We want to avoid getting too many reaching definitions → should start sets at \perp

Monday, November 14, 2011

Transfer functions

- Forward analysis, so need a slightly different formulation
 - Merged data flowing into a statement
$$IN(s) = \bigcup_{t \in pred(s)} OUT(t)$$
$$OUT(s) = gen(s) \cup (IN(s) - kill(s))$$
- What are *gen* and *kill*?
 - *gen*(s): the set of definitions that *may* occur at s
 - e.g., *gen*($s_1: x = e$) is $\langle s_1, x \rangle$
 - *kill*(s): all previous definitions of variables that are *definitely* redefined by s
 - e.g., *kill*($s_1: x = e$) is $\langle *, x \rangle$

Monday, November 14, 2011

Available expressions

- We've seen this one before
- What is the lattice? powerset of all expressions appearing in a procedure
- Forward or backward?
- Confluence operator?

Monday, November 14, 2011

Transfer functions for meet

- What do the transfer functions look like if we are doing a meet?

$$IN(S) = \bigcap_{t \in pred(s)} OUT(t)$$

$$OUT(S) = \mathbf{gen}(s) \cup (IN(S) - \mathbf{kill}(s))$$

- $\mathbf{gen}(s)$: expressions that *must be* computed in this statement
- $\mathbf{kill}(s)$: expressions that use variables that *may be* defined in this statement
 - Note difference between these sets and the sets for reaching definitions or liveness
- Insight: \mathbf{gen} and \mathbf{kill} must never lead to incorrect results
 - Must not decide an expression is available when it isn't, but OK to be safe and say it isn't
 - Must not decide a definition *doesn't* reach, but OK to overestimate and say it does

Monday, November 14, 2011

Analysis initialization

- Remember our formalization
 - If we start with everything initialized to \perp , we compute the least fixpoint
 - If we start with everything initialized to \top , we compute the greatest fixpoint
- Which do we want? It depends!
 - Reaching definitions: a definition that *may* reach this point
 - We want to have as few reaching definitions as possible \rightarrow use least fixpoint
 - Available expressions: an expression that was *definitely* computed earlier
 - We want to have as many available expressions as possible \rightarrow use greatest fixpoint
- Rule of thumb: if confluence operator is \sqcup , start with \perp , otherwise start with \top

Monday, November 14, 2011

Analysis initialization (II)

- The set at the entry of a program (for forward analyses) or exit of a program (for backward analyses) may be different
 - One way of looking at this: start statement and end statement have their own transfer functions
- General rule for bitvector analyses: no information at beginning of analysis, so first set is always $\{ \}$

Monday, November 14, 2011

Very busy expressions

- An expression is *very busy* if it is computed on *every path* that leads from a program point
 - Why does this matter?
 - Can calculate very busy expressions early without wasting computation (since the expression is used at least once on every outgoing path) – this can save space
 - Good candidates for loop invariant code motion

Monday, November 14, 2011

Very busy expressions

- Lattice?
- Direction?
- Confluence operator?
- Initialization?
- Transfer functions?
 - Gen? Kill?

Monday, November 14, 2011

Four types of dataflow

- Analysis can either be *forward* or *backward*
- Analysis can either be over *all paths* or over *any path*
 - All paths: merges consider values from all paths
 - Any path: merges consider values from any path

	All paths	Any path
Forward	available expressions	reaching definitions
Backward	very busy expressions	liveness analysis

- What kind of analysis is constant propagation?

Monday, November 14, 2011