Static analysis

December 15, 2005

Program analysis techniques

– Dataflow analysis
mainly techniques originating in compiler construction
emphasizes tradeoff betweeen precision and efficiency

– Constraint-based analysis
general framework for solving analysis problems by representing them as constraint relations between sets: generic and efficient algorithms

– Abstract interpretation
simplifies program by defining a semantics that considers only those aspects relevant for the desired property

– Type systems
by defining an appropriate type systems, many properties can be converted to type checking or type inference problems

Dataflow analysis

Techniques originating in the compiler domain
– used for code generation (e.g. register allocation)
– and code optimization (constant propagation, lifting common subexpressions, detecting unused variables, etc.)

Techniques have evolved and have been unified into a general framework applicable also to other code analysis problems.

Basic approach:
– construct the program control flow graph (CFG)
– observe how properties of interest change during program execution (upon traversing the nodes / edges of the CFG)

Program control flow graph

A representation in which:
– nodes are statements
– edges indicate sequencing of statements
⇒ we can have nodes with:
  – a single successor (straight-line code, e.g. assignments)
  – several successors (branch statements)
  – several predecessors (join after branching)

Alternative representation:
– nodes are program points
– edges are statements together with their effects

Notations

$G = (N, E)$: control flow graph (N : nodes; E : edges)
s : one program statement (node in the control flow graph)
entry, *exit* : program entry and exit points
in(s) : set of edges that have s as destination
out(s) : set of edges that have s as source
src(e), *dest*(e) : source and destination of edge e
pred(s) : set of predecessors of statement s
succ(s) : set of successors of statement s

With these notions, we write *dataflow equations* that describe how the analyzed values (dataflow facts) change from one statement to to the next

we use subscripts $_{in}$ and $_{out}$ for the value analyzed at entry and exit from statemens s.

Example: Reaching definitions

What are all definitions (assignments) that can reach the current program point? (before their assigned values are overwritten)

Elements of interest are pairs: (variable, source line of definition)
For each statement (identified by its label l) we are interested in the value before and after its execution: $RD_{in}(s)$ and $RD_{out}(s)$
– the initial node in the graph is not reached by any definition

$$RD_{out}(entry) = \{(v, ?) \mid v \in V\}$$

– an assignment $l : v \leftarrow e$ erases all previous definitions for variable v (but not for other variables) and introduces the current line (definition)

$$RD_{out}(l : v \leftarrow e) = (RD_{in}(s) \setminus \{(v, s')\}) \cup \{(v, l)\}$$

– definitions on entry of a statement are the union of definitions at exist of the predecessor instructions:

$$RD_{in}(s) = \bigcup_{s' \in pred(s)} RD_{out}(s')$$

Example: Live variables analysis

At each program point, what are the variables whose value will be used on at least one of the possible program paths from this point ?
(useful in compilers for register allocation)

Transfer function: $LV_{in}(s) = (LV_{out}(s) \setminus \mathit{write}(s)) \cup \mathit{read}(s)$
(a variable is *live* before s if it is read by s, or it is *live* after s without being written by s) $\Rightarrow$ direction of analysis is *backward*

Operation for combining values/joining paths (*meet*):

$$LV\_eout(s) = \begin{cases} \emptyset & \text{if } \mathit{succ}(s) = \emptyset \\ \bigcup_{s' \in \mathit{succ}(s)} LV\_ein(s') & \text{otherwise} \end{cases}$$

$\Rightarrow$ combination done by union (*may*, on at least one path)

Example: Available expressions

At each program point, what are the expressions whose values has been previously computed, without it having changed, on all paths to this point?
(if value is stored in a register, it need not be recomputed) Transfer function: $AE_{out}(s) = (AE_{in}(s) \setminus \{e \mid V(e) \cap \mathit{write}(s) \neq \emptyset\})$
$\cup \{e \in Subexp(s) \mid V(e) \cap \mathit{write}(s) = \emptyset\}$
(expressions on entry to s which have no variables modified by s, and any expressions computed at s without changes in their variables)

Combination operation (meet):

$$AE_{in}(s) = \begin{cases} \emptyset & \text{if } \mathit{pred}(s) = \emptyset \\ \bigcap_{s' \in \mathit{pred}(s)} AE_{out}(s') & \text{otherwise} \end{cases}$$

$\Rightarrow$ combination done by intersection (*must*, on all paths);
analysis is *before*

Example: Very busy expressions

What are the expressions which must be evaluated on any path from the current program point before the value of an appearing variable is modified ?
$\Rightarrow$ evaluation can be lifted to current point, before any branches
– a backward analysis, universaly quantified (*must*)

$$VBE_{in}(s) = (VBE_{out}(s) \setminus \{e \mid V(e) \cap \mathit{write}(s) \neq \emptyset\}) \cup Subexp(s)$$

$$VBE_{out}(s) = \begin{cases} \emptyset & \text{if } \mathit{succ}(s) = \emptyset \\ \bigcap_{s' \in \mathit{succ}(s)} VBE_{in}(s') & \text{otherwise} \end{cases}$$

Analyzed properties (dataflow facts)

Concretely: We might wish to analyze several properties, such as:
– value of a variable at a program point
– or the *interval* of values for a variable
– of sets of variables (live), expressions (available, very busy), possible definitions for a variable (reaching definitions), etc.

Abstractly: a set D of values for a property (*dataflow facts*)
Restriction: D is a *finite* set

Partially ordered sets

Concretely:
– we have associated with program points *sets* of values for the analyzed property
– we have iteratively recomputed the corresponding sets, by *union* or *intersection* operations, enlarging or restricting the set of values

What are the essential properties that allow this kind of calculation ?

Abstract: O *partially ordered set* $(L, \sqsubseteq)$ is a set equipped with a*partial order relation* $\sqsubseteq \subseteq L \times L$, i,e., a relation which is:
– reflexive, $x \sqsubseteq x$ for any $x \in L$
– transitive, $x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$, for any $x, y, z \in L$
– antisymmetric: $x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$, for any $x, y \in L$

Example: powerset $(\mathcal{P}(D), \subseteq)$ or $(\mathcal{P}(D), \supseteq)$

Lattices

(complete) lattice = a partially ordered set in which any finite subset has a least upper bound and a greatest lower bound.

l_0 is an upper bound of $Y \subseteq L$ if $\forall l \in Y$ we have $l \sqsubseteq l_0$
l_0 is a lower bound of $Y \subseteq L$ if $\forall l \in Y$ we have $l_0 \sqsubseteq l$

Denote: $\bigsqcup Y$: the least upper bound of the set $Y \subseteq L$
$\bigsqcap Y$: the greatest lower bound $Y \subseteq L$
şi $\bot = \bigsqcup \emptyset = \bigsqcap L \quad \top = \bigsqcap \emptyset = \bigsqcup L$

We define the operations
meet : $x \sqcap y = \bigsqcap \{x, y\}$
join : $x \sqcup y = \bigsqcup \{x, y\}$
(for powerset: intersection, union)

Lattices (cont.)

The operations $\sqcap$ (*meet*) and $\sqcup$ (*join*) are:
– commutative
– associative
– $x \sqcap \bot = \bot$ and $x \sqcup \top = \top$, for any x.

A *distributive* lattice: one in which the operators $\sqcap$ and $\sqcup$ are mutually distributive:
$x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$
$x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$

Transfer functions

Concretely: statements determine changes in the program state. The value of a variable after a statement is a function of its value at the beginning of the statement.

Abstractly: Each statement s has associated a transfer function $F(s) : L \to L$ that determines the way in which the value of the property at the beginning of the statment is modified by the statement:
$Prop_{out}(s) = F(s)(Prop_{in}(s))$ (forward analyses),
or conversely (backward analyses)

Restriction: we require transfer functions to be *monotone*

$$x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$$

(if we know more about the argument, we should know more about the result)

Particular case: *bitvector frameworks*: the lattice is a powerset $\mathcal{P}(D)$, transfer functions are monotone and of the form:

$$F(s)(v) = (v \setminus kill(s)) \sqcup gen(s)$$

(v = dataflow fact, $gen/kill(s)$ = information generated/deleted in s)

Dataflow equations

Example for forward analyses:

$$Prop_{out}(s) = F(s)(Prop_{in}(s))$$
$$Prop_{in}(s) = \prod_{s' \in pred(s)} Prop_{out}(s')$$

where we denote by $\prod$ the effect of combining information (*meet*) on several paths (could be $\cap$ or $\cup$)

Initially, we know the value $Prop_{out}(entry)$.

For backwards analyses, the roles of in and out change, and the value of $Prop_{in}(exit)$ is known.

Solution: *worklist* algorithm

To compute the solution for the above equation system, we use an iterative algorithm which propagates changes in the direction of the analysis.

foreach $s \in N$ **do** $Prop_{in}(s) = \top$ /* no info */
$Prop_{in}(entry) = init$ // depending of the analysis
$W = \{entry\}$
while $W \neq \emptyset$
  choose $s \in W$
  $W = W \setminus \{s\}$
  $Prop_{in}(s) = \prod_{s' \in pred(s)} Prop_{out}(s')$
  $Prop_{out}(s) = F(s)(Prop_{in}(s))$
  if change **then**
    forall $s' \in succ(s)$ **do** $W = W \cup \{s'\}$

Termination: fixpoint condition

Termination of the analysis is guaranteed if the transfer function is monotone: $x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$, which implies that computed properties change in a monotone way.

Def: *Fixpoint* for a function f: a value x for which $f(x) = x$
Tarski's Theorem guarantees that a monotone function over a lattice has a minimal and a maximal fixpoint.

The worklist algorithm computes the minimal fixpoint for the given system of transfer functions.

Meet over all paths

We wish to compute the combined effect of program statements:
for the sequence of instructions $p = s_1 s_2 \ldots s_n$ we define

$$F(p) = F(s_n) \circ \ldots \circ F(s_2) \circ F(s_1)$$

and we wish to compute:

$$\prod_{p \in Path(Prog)} F_p(entry)$$

But the worklist algorithm combines the effects at each meet before computing further. Since functions are monotone, we have:

$$f(x \sqcup y) \sqsupseteq f(x) \sqcup f(y)$$

thus the analysis *loses precision*
For *distributive* transfer functions we have equality: $f(x) \cup f(y) = f(x \cup y)$

It can be shown that the iterative worklist algorithm (the fixpoint solution) is equivalent with computing the solution by combining values over all possible paths (*meet over all paths*).
⇒ oombining the individual execution paths does not lose information

The examples given so far (live variables, etc.) are distributive

Classification of analyses

– forward or backward
– must or may
– control flow sensitive or control flow insensitive:
do we need to consider the order of statements in the program ?
– no: what variables are used/changed, what functions are called, etc.
– yes: properties effectively depending on values computed by the program
– context dependent or context independent
for programs with procedures: is the analysis of each procedure specialized depending on its call point, or is a single analysis (procedure summary) employed ?