Lecture notes – Introduction to abstract interpretation

August 29, 2019

Slide 3/18 – collecting semantics. A *state* is a snapshot of the memory of a program. In the simplest setting, a state is an element of of $Var \to Val$, where Var is the set of program variables and Val the set of all possible values. We use $State$ to denote the set of all possible states.

In case we have heap memory, pointers, record structures, etc., then states are more complex. In this setting *addresses* would need to be treated as values. Also, we would need to map each address (plus an offset, indicating a field) to a value. In general, any other resource that the program has access to (e.g., file pointers) would also need to be modeled. Furthermore, the structure of states can be extended, when required, to encode information that is not present in memory at run time; e.g., along with each address, one might track the statement number of the *last* statement that wrote to that address. In the rest of this lecture we stick to the simple notion of states outlined earlier.

Let I be the initial program point in a program, and let $\mathrm{SS}(I)$ be the set of initial states possible at I (this has to be given; in the slides we also refer to $\mathrm{SS}(I)$ as S_0). The *collecting semantics* of a program is a specification of the concrete meaning of the program. It is a map from program points to sets of states; i.e., each program point N is mapped to the precise set of all possible states $\mathrm{SS}(N)$ that can arise at the point N due to all paths that end at N when the program is run from any initial state in a given set of initial states $\mathrm{SS}(I)$.

Note that in general an infinite number of paths may end at a point N. Also, an infinite number of possible states may arise at N even due to a *single* path (e.g., due to non-deterministic functions such as odd() and even()). In general, it is impossible to compute the precise collecting semantics of programs.

In the example in the slide, each state is a pair of values, one for variable p and other for variable q. In this example it does not matter what the initial set of states $\mathrm{SS}(I)$ is, because p and q are both defined being used. Initially, we show the set of states that arise at the end of each path in the program. Finally, at each program point, we collect all the states that arise at that point due to all paths that end there. This is the collecting semantics.

Slide 4 – An abstract interpretation. The abstract interpretation is a 3-tuple, and is *given* to the analysis. The analysis designer needs to provide this. The analysis subsequently proceeds automatically using this.

Note that when we say that the lattice D needs to be given, we mean not only the elements of D, but also the definitions of the operators *meet* and *join* (actually, *join* alone is sufficient). The designer needs to provide a finite representation of the lattice (even if it is infinite), as well as some representation of the *join* operator.

The designer needs to provide transfer functions for each kind of statement and condition. Again, although the number of unique statements expressible in a language is infinite, the designer needs to give a finite representation of the transfer functions of all possible statements and conditionals. This is typically done by giving transfer functions for the individual operators in the language (see the Introduction slides), and then defining transfer functions for expressions recursively using the transfer functions of the individual operators.

The function γ is actually not used during analy-

sis. It is used only to express the correctness of the analysis (more on this later).

Slide 5 – example. This slide shows an abstract interpretation 3-tuple for odd-even analysis.

The lattice D is shown at the top of the figure. In the slide we show only the transfer function of an assignment statement. In general, the designer would also need to provide a transfer function for each branch of a conditional. For e.g.,

If MN is the "true" edge of the conditional "`p` == 2"

$f_{MN}(s) = (e, s[\mathrm{q}])$

If MN is the "true" edge of the conditional "`p` == `q`"

$f_{MN}(s) = \bot$, if s[p] is o and s[q] is e or vice versa
$(s[\mathrm{p}], s[\mathrm{p}])$, else if s[p] is o or e
$(s[\mathrm{q}], s[\mathrm{q}])$, else if s[q] is o or e
(oe, oe), otherwise

If MN is the "true" edge of the conditional "`p` == 2 || `q` == 3"

$f_{MN}(s) = s$

and so on.

In this slide the lattice, transfer functions, and γ map shown are for programs with two variables, named p and q. This raises the question, is an abstract interpretation only suitable for a class of programs with the same number of variables, same variables names, etc.? A "yes" answer would not be so good. Actually, what can be done is to design abstract interpretations that are *parameterizable*. That is, the analysis designer could somehow provide a generic abstract interpretation 3-tuple, and also a mechanical way to *instantiate* it for any given program. For simplicity we ignore this issue in our slides. The example abstract interpretation 3-tuples that we show are already instantiated for specific classes of programs.

Slide 6 – collecting abstract values. In this slide we illustrate the notion of *abstract* join-over-all-paths (i.e., abstract JOP) solution.

An initial abstract value d_0 at the entry of the program needs to be given. Let's take it as (oe, oe) in this example (again, because all the variables are defined before being used, it does not matter in this example what initial abstract value is used).

Initially, we show the abstract state (i.e., element of D) that arises at the end of each path. For e.g., the abstract state at the end of path ABC is $f_{BC}(f_{AB}(f_{IA}(d_0)))$, where I is the initial point. f_{IA} is the transfer function of the first assignment node; it returns (o, e) no matter what its argument is. f_{AB} and f_{BC} are the transfer functions of the merge node and the *true* branch of the condition node, respectively; both of these transfer functions happen to be *identify*. Therefore, the above computation yields (o, e).

Then, on the right side, we show against each program point the *join* of the abstract states at the end of all paths that end at that point. This is the abstract JOP (analogous to the collecting semantics).

Note that we have not yet talked about *how* to compute the abstract JOP. It is computable precisely only when the given abstract interpretation 3-tuple uses *distributive* transfer functions (more on this later).

Slide 7 – comparison of abstract JOP states and collecting states. Here, we show that at each program point the γ image of the abstract JOP is a superset of the corresponding collecting semantics (i.e., set of concrete states).

At most points, it is a strict superset, thus illustrating the fundamental fact that abstract interpretation is basically an approximate analysis.

Slide 8

Here we state the fundamental definition of when an abstract interpretation is said to be *correct*. Intuitively, for any program, at all program points, the γ image of the abstract JOP at that point needs to be a superset of the collecting semantics at that point.

Slide 9 – another example program. Shows collecting semantics for another example program.

Slides 10 – abstract interpretation 3-tuple for constant propagation. This slide shows the abstract lattice for the classical problem of constant propagation (used in compilers).

The lattice shown is for programs with two variables, x and y. The lattice is an infinite lattice (but of bounded height). Only a selection of elements of the lattice are actually shown in the figure.

Each element of the lattice, except $\perp$, is a *partial function* from variable names to values. For convenience we denote each partial function as a set of pairs, where each pair can be seen as a *constraint* on the value of the corresponding variable. Thus, e.g., $\emptyset$ indicates that neither of the variables is constrained. Using this notation the join of any two non-$\perp$ elements can be seen as set-intersection.

The given γ map is shown at the bottom of the slide.

In general, γ needs to *monotonic*. That is, for any two lattice elements d_1 and d_2 such that $d_1 \leq d_2$, $\gamma(d_1)$ needs to be a subset of $\gamma(d_2)$. Also, typically, $\gamma(\perp)$ is the empty set while $\gamma(\top)$ is the set of all concrete states.

Slide 11 – transfer functions

The definition of the transfer function in Slide 11 is self-explanatory. Again, the transfer functions for conditionals have been omitted (although they are very interesting).

Here are a few illustrations of the transfer function shown, for the statement $n \equiv$ "x := x + y":

$$\begin{array}{lcl} f_n(\perp) & = & \perp \\ f_n(\emptyset) & = & \emptyset \\ f_n((x,1)) & = & \emptyset \\ f_n(\{(x,1),(y,2)\}) & = & \{(x,3),(y,2)\} \end{array}$$

In general the transfer functions need to be monotonic.

Also, they need to satisfy the following correctness property: for any element $d_1 \in D$, if $f_n(d_1) = d_2$, then for every concrete state $s \in \gamma(d_1)$, $nstate_n(s)$ needs to be contained in $\gamma(d_2)$, where $nstate_n$ is the concrete semantics of the statement n.

It is in order to satisfy the above property that $f_n((x,1))$ has to return $\emptyset$. (No other lattice element exists such that it's γ image is a superset of the set $\{(i,j)|i = j+1\}$.)

Slides 12 and 13

Slides 12-13 are analogous to Slides 6 and 7. It is noted that the imprecision in the analysis is fundamentally due to the abstract lattice being an approximation of the concrete domain (which is $(2^{State}, \subseteq)$).

In other words, imprecision results because the abstract lattice cannot make all distinctions that the concrete domain can, even though we use the best possible abstract transfer functions for this abstract lattice. (This same fact holds in the example in Slide 7 also.)

This imprecision is the price to pay for attempting to solve an undecidable problem.

Slides 14-16. These slides contain formalism, and are mostly self-explanatory.

A few comments on Slide 15: For the conditional node, we will need two transfer functions, f_{LM} and f_{LN}. Similarly, for the join node, we will need two transfer functions f_{KM} and f_{LM} (both being *identity*).

Slide 17. The definition of the function *nstate* falls out of the concrete semantics of expressions and assignments as specified by the programming language.

A natural thought is that *nstate* should have the signature $State \rightarrow State$. However, our formulation allows a statement to yield *no states* on certain inputs (e.g., on inputs that cause a divide-by-zero exception). Also, it supports non-deterministic statements that return one among a set of possible output states for a given input state, e.g., an input statement or a call to a random number generator. For any state s, during actual runtime, any execution of the node/branch that is in between program points M and N using s as the incoming state should be guaranteed to produce an outgoing state that is an element of $nstate_{MN}(s)$.

Note that if L is the program point that precedes a conditional node, and if M and N are its two successor program points, then, provided the condition checked in the node is deterministic, for any concrete state s, s would be present in at most one of the sets $nstate_{LM}(s)$ or $nstate_{LN}(s)$. It is possible that s is present in neither set (this will happen if evaluation of the condition on the incoming state s causes the conditional node to terminate the execution of the program).