Abstract Interpretation

a first introduction

Hanbing Liu

hbl@cs.utexas.edu

University of Texas at Austin

Abstract Interpretation

Programs denote computations in some universe of objects.

Abstract Interpretation

Programs denote computations in some universe of objects.

A same program can be interpreted differently with respect to different universes of objects.

Abstract Interpretation

Programs denote computations in some universe of objects.

A same program can be interpreted differently with respect to different universes of objects.

For example: $-1515 * 17$ may denote a computation on the abstract universe of $\{(+), (-), (+/-)\}$:

$$-1515 * 17 \Rightarrow -(+) * (+) \Rightarrow (-) * (+) \Rightarrow (-)$$

$$-1515 + 17 \Rightarrow -(+) + (+) \Rightarrow (-) + (+) \Rightarrow (+/-)$$

Abstract Interpretation

Programs denote computations in some universe of objects.

A same program can be interpreted differently with respect to different universes of objects.

For example: $-1515 * 17$ may denote a computation on the abstract universe of $\{(+), (-), (+/-)\}$:

$$-1515 * 17 \Rightarrow -(+) * (+) \Rightarrow (-) * (+) \Rightarrow (-)$$

$$-1515 + 17 \Rightarrow -(+) + (+) \Rightarrow (-) + (+) \Rightarrow (+/-)$$

One wants to study some aspects of the concrete (but often more complicated) executions by studying corresponding properties of abstract (thus simpler) executions.

Abstract Interpretation

An abstract interpretation is defined as a non-standard (approximated) program semantics obtained from the standard (or concrete) one by replacing the actual (concrete) domain of computation and its basic (concrete) semantic operations with, respectively, an abstract domain and corresponding abstract semantic operations.
`http://www.doc.ic.ac.uk/~herbert/epsrc/node2.html`

Abstract Interpretation

The abstract interpretation research studies:

- when two interpretations are related — when and what kind of properties derived in one interpretation can be accepted as properties of another interpretation.
- how to construct an abstract interpretation of a program, that is both
  - “simple” — finite (?)
  - “useful” — sufficiently accurate (?)

Background

- Motivation
  - It relates to my work of verifying the bytecode verifier
  - It provides a unified way for looking at various problems: code optimization, modeling checking,
  - Formulating the abstract interpretation concept rigorously is interesting
- Objective of this short talk
  - Concepts of *a.i.*, *c.i.*, *safe simulation*
  - Conditions for ensuring *safe simulation*
  - Tricks for getting "finite" *a.i.*
- Plan

Traces as “generic” program semantics

- Standard (concrete) semantics: concrete execution traces, where traces are sequences of states.

  For example, meaning of a Java bytecode program

  We will see a concrete trace example for a flowchart program later.

Traces as “generic” program semantics

- Standard (concrete) semantics: concrete execution traces, where traces are sequences of states.

  For example, meaning of a Java bytecode program

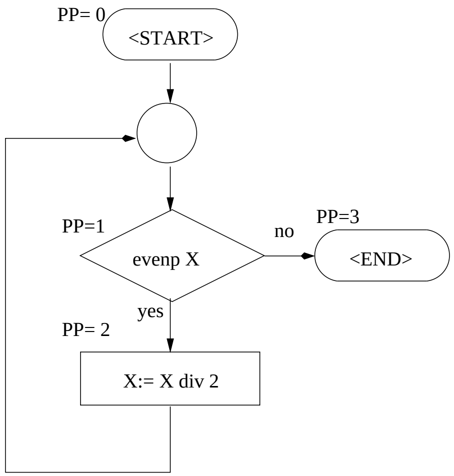
  We will see a concrete trace example for a flowchart program later.

- Abstract semantics: traces are (possibly) trees of transitions that connect abstract states.

  For example, abstract semantics of Java bytecode programs.

Flowchart program example

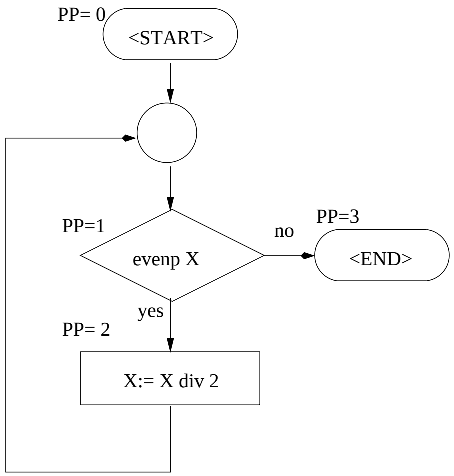
- Flowchart program syntax:



Programs are constructed from a set of nodes; nodes have predecessors and successors; type of nodes: test nodes, junction nodes, assignment nodes; nodes mention identifiers and expressions

Flowchart program example

- Flowchart program syntax:

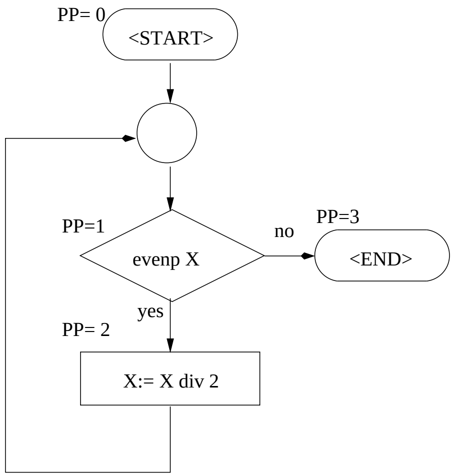


Programs are constructed from a set of nodes; nodes have predecessors and successors; type of nodes: test nodes, junction nodes, assignment nodes; nodes mention identifiers and expressions

- Universe of objects: numbers; mappings from identifiers to numbers; program points

Flowchart program example

- Flowchart program syntax:



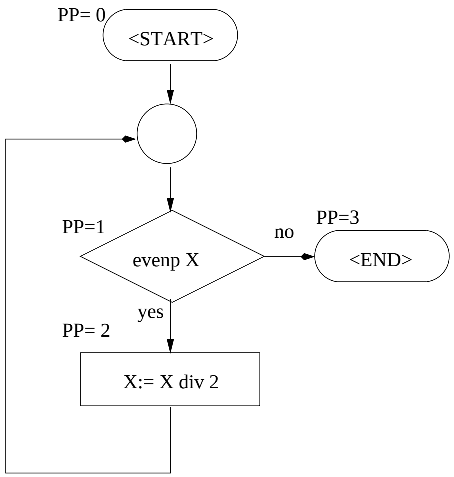
Programs are constructed from a set of nodes; nodes have predecessors and successors; type of nodes: test nodes, junction nodes, assignment nodes; nodes mention identifiers and expressions

- Universe of objects: numbers; mappings from identifiers to numbers; program points

For rigorous descriptions, see Cousots' paper: *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Program by Construction or Approximation of Fixpoints*

Flowchart program example

- Concrete interpretation as a sequence of states:

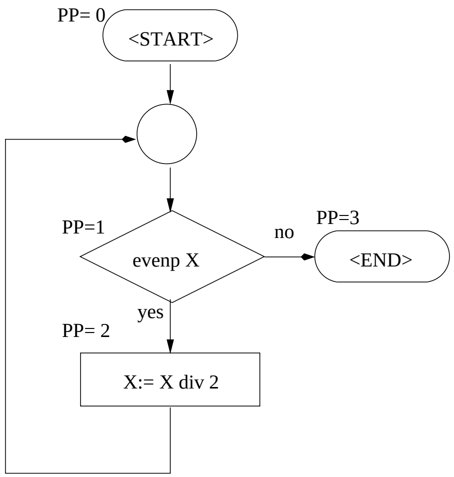


Suppose for a concrete execution starting from x == 12:

12@pp=1→12@pp=2→6@pp=1→6@pp=2→3@pp=1 →3@pp=3

Flowchart program example

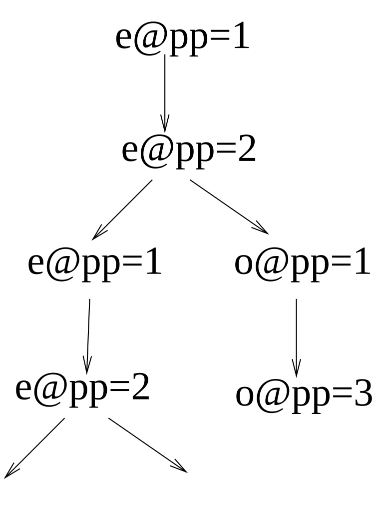
- Concrete interpretation as a sequence of states:



Suppose for a concrete execution starting from x == 12:

12@pp=1→12@pp=2→6@pp=1→6@pp=2→3@pp=1 →3@pp=3

- Abstract interpretation as a tree of abstract states



On universe of {e,o}, div operation is interpreted to produce both e, o.

Execution from x == e is an infinite tree

Safe simulation

- A value c being safely approximated/represented by a:

$$c \; Safe_{val} \; a$$

  where binary relation $Safe_{val} \subseteq Val \times AbsVal$

- Similarly, a $Safe_{state}$ relation between concrete states and abstract states

$$c \vdash pp \; Safe_{state} \; a \vdash pp \text{ iff } c \; Safe_{val} \; a$$

- $Safe_{trace}$ definition:
  $t_c \; Safe_{trace} \; t_a$ iff $root(t_c) \; Safe_{state} \; root(t_a)$ and for every transition, $root(t_c) \rightarrow t_{c_i}$, there exists a transition, $root(t_a) \rightarrow t_{a_j}$, and $t_{c_i} \; Safe_{trace} \; t_{a_j}$

“Fundamental theorems”

The subset relation between reachable states

$$coll_c(pp) \subseteq \gamma(coll_a(pp))$$

where $coll_t(pp)$ = { v | $v \vdash pp$ is a state in trace t }; and where $\gamma(S)$ = { c | $\exists a \in S$ such that $c\ Safe_{val}\ a$}, $S \subseteq AbsVal$

More generally, for certain logic L (e.g. *box-mu-calculus*), interpretable on t_c and t_a, one may prove

> $t_c\ Safe_{trace}\ t_a$ implies for all formula $\phi \in L$,
> $t_a \models \phi \Rightarrow t_c \models \phi$.

Safe simulation: another formulation

An alternative formulation:

There exists a $\beta : Val \rightarrow AbsVal$:
For all program points, pp, and $c \in Val$,
$c \vdash pp \rightarrow_c c' \vdash pp'$ implies there exists $a' \in AbsVal$
such that $\beta(c) \vdash pp \rightarrow_a a' \vdash pp'$ and $\beta(c') \sqsubseteq a'$

We also require that transition relation in the abstract interpretation is *monotonic* with respect to the *approximation ordering*, $\sqsubseteq$:

$a \vdash pp \rightarrow_a a' \vdash pp'$ and $a \sqsubseteq b$ implies
$b \vdash pp \rightarrow_a b' \vdash pp'$ and $a' \sqsubseteq b'$

We define $Safe_{val}$ as:

$c \; Safe_{val} \; a$ if $\beta(c) = a$, or $\exists a' \in AbsVal, a' \sqsubseteq a$ and
$\beta(c) = a'$

Construct "finite" a.i.

One wants *a.i.* to be finite (a trace can be infinite but needs to be a regular tree), so that it can be explored effectively. Common technique is to approximate with "memorization":

> If a node is $v \vdash pp$, it is generalized to $v \sqcup v' \vdash pp$, where $v' =$ { v | $v \vdash pp$ appears earlier in the trace }

For this to produce a regular tree that finitely represents *a.i.*, we need $AbsVal$ be partially ordered and has finite-chain property.

Summary

One is motivated to show abstract interpretation of a program being a safe simulation because:

- "Fundamental theorems" about safe simulation
- Abstract interpretation is simpler

To establish a safe simulation, one approach is to:

- Define β that maps concrete state into abstract state
- Show that transition relations on the abstract domain is "monotonic"
- Show that for any possible concrete transition, there is a corresponding transition on the abstract domain

Java bytecode verification

Java bytecode verification

One of my goal is to relate properties asserted on abstract executions to properties of concrete executions.
In particular, I need to show

- $coll_c(pp) \subseteq \gamma(coll_a(pp))$
  That is the set of reachable states $coll_c(pp)$ is subset of the set of states which correspond to reachable abstract states $coll_a(pp)$
- Furthermore, I need to show that the bytecode verifier checking on the abstract states implies the runtime checking on any of the corresponding concrete state

  That is if the abstract execution does not enter an error state, concrete executions from those corresponding abstract state will not enter an error state.

BCV as *a.i.*

Roughly:

- β: frame-sig
- $\sqsubseteq$ on $AbsVal$:
  sig-frame-more-general
- Monotonicity:
  - (bcv::check-* gframe) $\Rightarrow$ (bcv::check-* sframe)
  - (bcv::execute-* sframe) $\sqsubseteq$ (bcv::execute-* gframe)
- $c \vdash pp \rightarrow_c c' \vdash pp'$ implies there exists $a' \in AbsVal$ such that $\beta(c) \vdash pp \rightarrow_a a' \vdash pp'$ and $\beta(c') \sqsubseteq a'$
  - (bcv::check-* (frame-sig s)) $\Rightarrow$ (djvm::check-* s)
  - (frame-sig (djvm::execute-* s))
    $\sqsubseteq$ (bcv::execute-* (frame-sig s))

Frame-sig

Built upon value-sig:

```
(defun value-sig (v cl hp hp-init curMethodPtr)
   (if (REFp v hp)
      (if (NULLp v)
          'null
        (let ((obj-init-tag (deref2-init v hp-init))
              (obj (deref2 v hp)))
          (if (not (consp obj-init-tag))
              (fix-sig (obj-type obj))
            (if (equal (cdr obj-init-tag) curMethodPtr)
                ;; if the object is created in this method
                ;; then translate into an uninitialized(Offset)
                (cons 'uninitialized (car obj-init-tag))
              'uninitializedThis))))
    (tag-of v)))
```

AALOAD

BCV and DJVM's check-AALOAD

```
(defun check-aaload (inst env curFrame)
  (declare (ignore inst))
  (mylet* ((ArrayType (nth1OperandStackIs 2 curFrame))
           (ElementType (ArrayElementType ArrayType)))
    (validtypetransition env
         '(int (array (class "java.lang.Object")))
          ElementType

          curFrame)))


(defun AALOAD-guard (inst s)
  (mylet* ((index (safe-topStack s))
           (array-ref (safe-secondStack s)))
    (and (consistent-state s)
         (topStack-guard-strong s) ....
         (<= (len (operand-stack (current-frame s))) (max-stack s))
         (or (CHECK-NULL array-ref)
             (and (CHECK-ARRAY-guard (rREF array-ref) (heap s))
               (not (primitive-type? (array-component-type (obj-type
```

Lemma

Show AALOAD.lisp