Static analysis project

Semantics and Application to Program Verification

Antoine Miné

École normale supérieure, Paris

year 2015–2016

Course project

14 April 2016

Organization

Goal: program a simple static analyzer.

You can work alone, or in groups of 2.

You must provide:

- the source code of your analyzer, with a Makefile
- a small (5-6) set of example sources to analyze, with the results given by your analyzer
- a small report (1–2 pages) discussing your analyzer, your experience, and your experiments

Project description, documentation and source material at:

https://www-apr.lip6.fr/~mine/enseignement/l3/2015-2016/project

Analyzer organization

Three parts:

- **Front-end:** given
  - parses a small C-like language
    integers, expressions, if-then-else, loops, gotos, functions
  - transforms it into a control-flow graph
- **Iterator:** must be implemented

  Worklist algorithm to propagate invariants in the graph.
  For intervals, iterations with widening in case of cycles (loops).
- **Abstract domains:** must be implemented
  - constant domain
  - interval domain

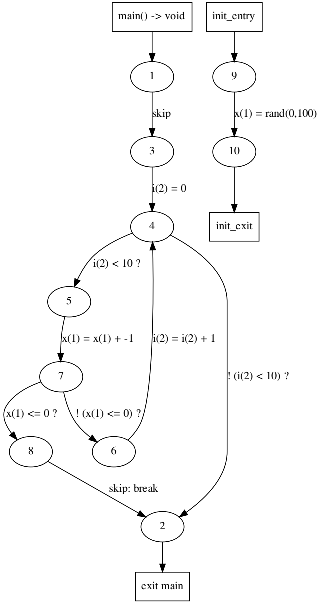
you can also start with a concrete domain, as in the lab sessions,
to develop and test your iterator before developing abstract domains

Front-end

```
int x = rand(0,100);

void main() {
  int i;
  for (i=0;i<10;i++) {
    x--;
    if (x<=0) break;
  }
}
```

- node: program locations
- arcs: instructions & control-flow
- initialization sub-graph
  plus one sub-graph per function
- variables are disambiguated
- expressions are simplified



Abstract domains

Suggested signatures (see the `mli` files in the project archive)

DOMAIN: $\mathcal{P}(\texttt{var} \to \mathbb{R})$

```
type t
init:   var list -> t
assign:   t -> var -> iexpr -> t
guard:   t -> bexpr -> t
join:   t -> t -> t
widen:   t -> t -> t
subset:   t -> t -> bool
```

VALUE_DOMAIN: $\mathcal{P}(\mathbb{R})$

```
type t
const:   Z.t -> t
unary:   t -> op -> t
binary:   t -> t -> op -> t
compare:   t -> t -> op -> t * t
bwd_unary:   t -> op -> t -> t
bwd_binary:   t -> t -> op -> t -> t * t
join:   t -> t -> t
widen:   t -> t -> t
subset:   t -> t -> bool
```

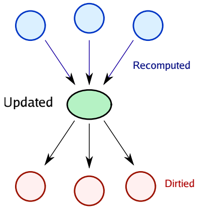
- assignment: bottom-up evaluation on expression trees
  (variables, `const`, `unary`, `binary`)
- guard: top-down refinement
  (`compare`, `bwd_unary`, `bwd_binary`)
- join, widening, subset: point-wise on each variable

More information in the following courses and lab sessions.

Iterator

Suggestion: worklist algorithm

- assign an abstract environment to each node
- keep a **worklist** of dirty nodes
- update a dirty node by recomputing all arcs going into the node and taking the join
- if unstable, add successor nodes to the worklist
- select widening points to break each cycle in the graph (accelerate convergence: put unstable interval bounds to $\pm\infty$, see next course)



Output:

- abstract invariant at each graph node
- list of assertion instructions that fail

Different from the method by interpretation on the abstract syntax tree, seen in the lab sessions!

Extensions

One extension to do, chosen among the following possible:

- backward analysis
  (from an assertion failure up to its cause)
- inter-procedural analysis (no recursivity)
  (flow from call sites to function entry, from function exit to return site)
- polyhedral analysis
  (using the Apron library)
- disjunctive analysis
  (using state partitioning, disjunctive completion, or trace partitioning)
- an extension of our own choice,
  after discussing it with the teacher!

These topics will be taught in the following courses.
More information on the web-page.