From Soundiness to Soundness

Yannis Smaragdakis
University of Athens

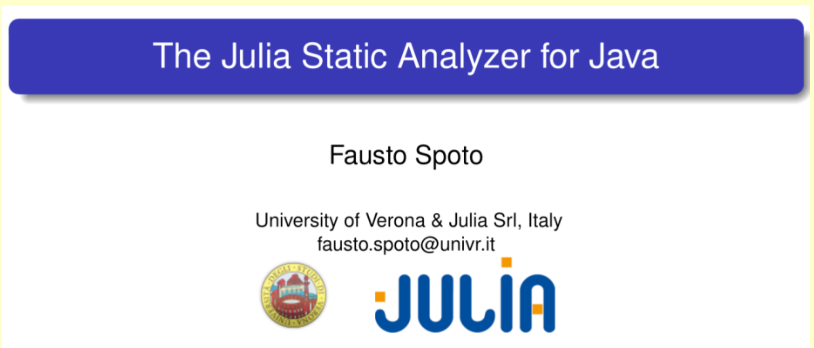
European Research Council
Established by the European Commission

OPERATIONAL PROGRAMME EDUCATION AND LIFELONG LEARNING
investing in knowledge society
MINISTRY OF EDUCATION & RELIGIOUS AFFAIRS, CULTURE & SPORTS
MANAGING AUTHORITY
European Union
European Social Fund
NSRF 2007-2013
EUROPEAN SOCIAL FUND
Co-financed by Greece and the European Union

LOGICBLOX

Soundness

- An oft-used term in program analysis
- Example quotes in recent keynote:



  - “A parallel library for the static analysis of Java bytecode”
  - “based on abstract interpretation”
  - “hence sound”

Define Soundness!

- What does it mean for an analysis to be ***sound***?
  - either a static one or a dynamic one

Sound =
"It works well" ?

Sound =
"It has a theory behind it" ?

***Sound* =**

“There is a proof of some property” ?

No!

- Soundness has a well-defined meaning
- It only has to do with the analysis itself
  - not with what we can prove about it
- Sound = "analysis claim implies truth"
- Same definition as in mathematical logic:
  - proof of P implies P
  - often:
    "the logic can only prove true theorems"

***Sound* =**

AnalysisClaim(P) → P

Examples

- Analysis: the program has a race → the race is real ("no false positives")
- Analysis: the program is well-typed → no run-time type errors ("no false negatives")
- Analysis: call may invoke these N methods → no others ever called ("overapproximate")
- Analysis: expressions must be aliases → they can never have different values ("underapproximate")

Hold on! You Just Told Us Soundness Means 4 Things?

- Yes! And that's the first difficulty
  - sound may mean "underapproximate", but also "overapproximate"
  - sound may mean "no false positives", but also "no false negatives"
- ***Sound* = AnalysisClaim(P) → P**
- But what claim does an analysis make?
- Often only in the mind of its user:
  claim is a matter of interpretation

Example Analysis Claims

- An analysis returns *x* results
  - is it a claim that these are the **only** ones?
    - a "*may-analysis*"
  - is it a claim that **at least** these are valid?
    - a "*must-analysis*"
- An analysis warns of bugs
  - is it a claim that these are **real** bugs?
    - a "*bug-detector*"
  - is it a claim that **no other** bugs exist?
    - a "*verifier*"

Common Patterns for *Correctness* Analyses

- Dynamic analyses are usually bug detectors
  - i.e., analysis claims to find bugs
  - sound = only true warnings
  - e.g., race detection, fuzzing, dynamic-symbolic execution
- Static analyses are often verifiers
  - analysis certifies the absence of errors
  - sound = finds all errors
  - e.g., type systems, data-flow analyses

What About Other Analyses?

- In the static analysis world:
  - *may/possible*-analysis = aims to be overapproximate
    - sound = all actual behaviors are captured
  - *must/definite*-analysis = aims to be underapproximate
    - sound = only captures actual behaviors

Now "Complete"

- We saw: ***Sound* = AnalysisClaim(P) → P**
- ***Complete* = P → AnalysisClaim(P)**

- ***Sound* =**
  AnalysisClaim(P) → P ≡
  ¬P → ¬AnalysisClaim(P) ≡
  ¬P → AnalysisClaim(¬P)
  - An analysis that is sound for a property P is complete for property ¬P, and vice versa
    - e.g., a sound verifier is a complete bug finder

Soundness In Static Analysis

- There is **no** practical static *whole-program* may-analysis that is sound
  - (whole-program: models the heap)
  - this is remarkable!
- What about all these soundness proofs, claims, etc.?
  - proof/claim is for a limited language
  - unsoundness is due to highly dynamic features in full language:
    reflection, dynamic loading, setjmp/longjmp, eval

Soundiness [CACM'15]

- *Soundy* analysis:
  - sound handling of most language features
  - deliberately unsound handling of a feature subset
    - subset well recognized by experts
- A soundy analysis aims to be as sound as possible without compromising precision and/or scalability
- All "sound" analyses are really just soundy

Why Is Soundness Difficult?

- `x = y.f;`
  `z = y.f;`
  - `x == y?`
  - y may have escaped to other thread
- `w.foo();` // only one `foo` in the program
  - is it the one called? Maybe more loaded dynamically
- `c = Class.forName(str);`
  - should it return all possible classes? Too imprecise

Why Is Soundness Difficult?

- Best-effort handling of complex features is too expensive!
- Different analysis logic: cannot just enumerate values
- More than half of the program non-analyzable
- Expensive: work wasted (more on this later)

- So, what can we do?

Approach I: Empirical Soundness

- *Empirical soundness*:
  - quantify unsoundness, get it close to zero
- It now makes sense to talk about "more sound" and "less sound"
- Try to capture practical usage patterns of dynamic language features
- Common theme in much recent work
  - Livshits et al. (JavaScript analysis for libraries)
  - Li et al. (Java reflection analysis)

Analysis Pattern: Inter-Proc. Back-Propagation [APLAS'15]

- Create dummy objects, see how they are used, determine what they could have been!
- ```
  Class c = Class.forName(className);
  …
  Object o = c.newInstance();
  …
  e = (Event) o;
  ```
  - c points to a *special* object, propagates as-if normal
  - when it gets to the cast, we can guess what c was
  ```

Analysis Pattern: Inter-Proc. Back-Propagation

- The same idea applies to lots of patterns

- ```
  Class c = Class.forName(className);
  …
  Field f = c.getField(fieldName);
  ```
  - when `c` gets to `getField`, we can guess what it was
    - if we know (something about) `fieldName`
  ```

Notes on Inter-Proc. Back-Propagation

- It is “more sound” to over-guess objects based on use
  - the analysis is a *may-analysis*
- Livshits et al. and Li et al. do the same but for fewer patterns, mostly intra-procedurally
  - why? To avoid over-guessing for reasons of *precision* and *analysis cost*
- We handle these concerns separately

Approach II: Full Soundness, for Parts of the Program

- Accept that a sound analysis will only give results for parts of the program, see how much
- *Defensive analysis*: sound-by-definition static analysis
- Anything that is inferred is guaranteed conservative (overapproximate)
- Need special encoding: a top value (T) to designate “any value”
- Need special handling to avoid wasting work

Wasting Work

- ```
  while (...)
  { x = y.fld;
    x.foo(y); }
  ```
- Say we know all the (currently) possible values of `y` *and* of `y.fld`
- We get values for `x`
- One of these results in a new `foo` target
- Yielding a T for y.fld
- This should invalidate all earlier values for `x`
  ```

Defensive Analysis

- ```
  while (...)
  { x = y.fld;
    x.foo(y); }
  ```
- Never infer anything unless guaranteed to have all values
- Values of `y` *and* of `y.fld` remain "unknown"
- Defensive: "unknown" and "all values" (T) are equivalent
- Idea: represent both by the *empty* set of values
  ```

Empty Set

- An empty set of values means “cannot bound”
- Lots of advantages:
  - no explicit representation, no cost
  - naturally encodes defensive behavior
    - no difference between “cannot be certain the set of values is bounded” and “the set of values is unbounded”
  - no wasted work: sets start empty and only grow
    - never *revert* to empty

Defensive Analysis: in Doop

- Datalog-based analysis framework for Java
  [OOPSLA'09, PLDI'10, POPL'11, OOPSLA'13, PLDI'13, PLDI'14, SAS'16, …]
- 2-3K logical rules (20-25KLoC)
- Very high performance (often 10x over prior work)
- Sophisticated, very rich set of analyses
  - subset-based analysis, fully on-the-fly call graph discovery, field-sensitivity, context-sensitivity, call-site sensitive, object sensitive, thread sensitive, context-sensitive heap, abstraction, type filtering, precise exception analysis
- High completeness: full semantic complexity of Java
  - jvm initialization, reflection analysis, threads, reference queues, native methods, class initialization, finalization, cast checking, assignment compatibility

http://doop.program-analysis.org

Defensive Analysis Results

- Can still cover ~40% of realistic programs
- Meaning: 40% of the program variables get sets of values that are not empty
- The rest conservatively over-approximated to empty, i.e., T

Conclusions

Recap

- Soundness is a property of an analysis
  - not a meta-property, nothing to do with proofs
- One should be clear on analysis “claims”
  - they are subject to interpretation, affect soundness
- No practical static analysis* is sound
  - surprising but true
- Once we accept this, we can do interesting stuff in this space
  - empirical soundness + defensive (lower coverage)