On quantifying the degree of unsoundness of static analyses

Dimitris Vardoulakis (dimvar@google.com)

Off the Beaten Track 2018

Static-analysis research

Assume soundness, then try to improve precision and scalability.

Static analyses in industry

Must be practical/useful. Often sacrifice soundness to achieve that.

E.g., trade soundness for:

- Precision, low number of false positives
- Scalability
- Simplicity of implementation

Examples of unsound analyses

ProGuard removes dead code but cannot detect reflective use. Programmers must specify explicitly which classes must be kept due to reflection.

TypeScript performs refactoring based on unsound type information.

Closure Compiler performs code optimizations based on unsound type information.

Unsoundness in JavaScript type systems

The main JS type systems (Closure Compiler, TypeScript, Flow) are unsound.

Hard-to-analyze JS features:

- Computed property accesses
- Adding/removing properties from an object at runtime
- Prototypal inheritance
- And many more[1]

[1] The design of JavaScript type systems (StrangeLoop 2017): https://www.youtube.com/watch?v=MuC8I1JBKv0

Evaluating an unsound analysis

Empirical evaluation based on user feedback.
Does it find the bugs people expect it to find?
Does it have many false positives?

Works well in practice, but no formal guarantees.
Comparing various analyses is very subjective.
Neglected in research literature.

Evaluating an unsound analysis

In addition to the empirical evaluation, can we evaluate unsound analyses in a mathematically rigorous way?

Potential benefits:

- Quantify “how unsound” an analysis is
- Compare different analyses more systematically
- Stimulate research on unsound analyses

In practice, soundness is a continuum

From the trivially unsound analysis to the perfect analysis, many in between.
How to formalize this intuition?

Consider a ML classifier that recognizes email spam. It is useful even if it misses some spam and classifies some non-spam as spam.

Use measures like precision and recall to evaluate static analyses.
A statistical view of program analysis instead of a logical view.

Precision and recall

Suppose we have a dataset of emails where we know which ones are spam and which are not.

Precision: of the emails we classify as spam, what percentage truly is spam?

Recall: of all spam emails in the dataset, what percentage do we detect?

High precision means few false positives.
High recall means few false negatives.

By using recall we can talk about degrees of soundness; we can say that an analysis is more sound than another analysis.

Using a labeled dataset to evaluate static analyses

ImageNet[1]: public dataset to evaluate image-recognition systems.
Millions of images, thousands of categories.

Create similar datasets to evaluate static analyses.

The type checker in Closure Compiler has ~2400 unit tests, ~100 diagnostic types.
INVALID_ARGUMENT_TYPE, WRONG_ARGUMENT_COUNT, MISSING_PROPERTY, *etc.*

Use the same diagnostic types across type checkers.
Use test suites to create a labeled dataset (first iteration).

[1] https://qz.com/1034972/the-data-that-changed-the-direction-of-ai-research-and-possibly-the-world

Using a labeled dataset to evaluate static analyses

Run the analysis on each program in the dataset:

- Finds an expected warning (true positive)
- Finds an unexpected warning (false positive)
- Misses an expected warning (false negative)
- Correctly finds no warnings (true negative)

Using a labeled dataset to evaluate static analyses

Can use TP, FP, TN, FN to calculate a score on the whole dataset.

A bit tricky.
A program can have several warnings, some of the same diagnostic type.
The analysis may find a TP and a FP for the same program.
Location of the warning matters as well.

ImageNet Challenge: object localization[1]
Calculate error for each image (between 0 and 1) based on the categories found, and overlap of bounding boxes.
Minimum average error across all images wins.

[1] http://image-net.org/challenges/LSVRC/2017/

Using a labeled dataset to evaluate static analyses

A quantitative way to compare analyses.

Implementation independent.
Formalizing the algorithm itself much more difficult.

A sound analysis becomes just another point in the design spectrum.
Soundness (perfect recall) not the best choice for some problems, due to low precision (*e.g.*, JavaScript type checking).

Instrumented interpreter as the source of truth

Getting labeled data not always feasible.
Too expensive/time consuming, working on an experimental language, *etc*.

Use an instrumented interpreter as the source of truth.
Compare the analysis result to the actual execution of a program.

Instrumented interpreter as the source of truth

Define abstraction function: 5 → {number}, true → {boolean}, *etc*.

Execute program, collect all values that flow through each program point.
Abstract these to get set *S*.
Run the analysis and compute a different set *S'* for the same program point.

- If $S = S'$, the analysis computed the correct result.
- If $S \subseteq S'$, the analysis overapproximated (false positive).
- If $S \supseteq S'$, the analysis underapproximated (false negative).
- Else, neither overapproximation nor underapproximation.

Compute the percentage of correct results, false positives, and false negatives for the whole program.

Instrumented interpreter as the source of truth

To evaluate the analysis on the whole language:

- If access to a large real-world codebase, can repeat this process for all programs in the codebase.
- Or, can generate many random programs and do the comparison (*e.g.*, for experimental language).

How to best deal with incomplete or dead code (*e.g.*, code that takes inputs, library code).

Caveats

We can use familiar techniques from the design of sound analyses for unsound analyses as well.

Not proposing to expand the use cases for unsound analyses, just to evaluate them more formally.

Difference with machine learning: static-analysis algorithms are fixed; their result does not change based on the training data.

Key takeaways

Shift in perspective: soundness as a spectrum; wide range of unsound analyses. Soundness not special, just another design point.

Unsound analyses can be evaluated rigorously: taking a cue from machine learning, develop measures that account for both false positives and false negatives.

Thank you!

Closure Compiler: property disambiguation

```
class Foo {
  Foo() { this.a = 1; }

  setprop() { this.a++; }
}

class Bar {
  Bar() { this.b = 2; }

  setprop() { this.b--; }
}

var x = new Foo;
x.setprop();
var y = x.a + (new Bar).b;
```

Closure Compiler: property disambiguation

```
class Foo {
  Foo() { this.Foo$a = 1; }

  Foo$setprop() { this.Foo$a++; }
}

class Bar {
  Bar() { this.Bar$b = 2; }

  Bar$setprop() { this.Bar$b--; }
}

var x = new Foo;
x.Foo$setprop();
var y = x.Foo$a + (new Bar).Bar$b;
```