

Semantic Solutions to Program Analysis Problems

Sam Tobin-Hochstadt

David Van Horn

PRL, Northeastern University

Abstract

Problems in program analysis can be solved by developing novel program semantics and deriving abstractions conventionally. For over thirty years, higher-order program analysis has been sold as a hard problem. Its solutions have required ingenuity and complex models of approximation. We claim that this difficulty is due to premature focus on *abstraction* and propose a new approach that emphasizes *semantics*. Its simplicity enables new analyses that are beyond the current state of the art.

Current Thoughts, New Ideas

Higher-order program analysis has been an important and recurring topic at PLDI, starting with Shivers' seminal paper [1] and continuing through the present [2]. However, past approaches are limited in the language features they can handle, require intricate formal models that are difficult to develop, verify, and maintain, and do not scale to new questions that we need to answer of programs. We propose a new approach in which interesting *analyses* can be developed by first developing interesting *semantics* and then using known techniques to approximate as a final step.

As an example, Meunier, *et al.* [3] develop a modular program analysis for higher-order behavioral software contracts. Meunier gives an analysis in the form of a large constraint set system and, separately, a dynamic reduction semantics. An important drawback is the dissimilarity between the semantics and the analysis. Both are complicated for the sake of establishing a correspondence, which is accomplished by shoehorning the semantics into an analysis, and tweaking to achieve modularity. Despite these efforts, the soundness theorem does not hold. Worse, the system was then abandoned, as it could not be maintained, extended, or implemented.

In contrast, we have taken the semantics of Meunier's language and *systematically derived* a similar whole-program analysis based on an abstract machine for the language [4]. The machine itself is derived from the semantics through known techniques, making its correctness proof straightforward. This step is purely a semantic refactoring; it has nothing to do with approximation. The machine, however, is in a form that abstracts naturally and transparently [5].

What remains is to make this analysis *modular*, enabling reasoning about programs that are missing some of their components. We solve this problem purely on the semantic side of the equation by extending the dynamic semantics with reductions for programs with missing components. Missing components are regarded as their contracts, which are given reduction rules corresponding to the reductions that may be taken by *any* value satisfying those contracts. As an example, consider the following program fragment consisting of two modules with unknown implementations, `keygen` and `rsa`, and a call to `rsa` to encrypt a string using a key from `keygen`. Inputs and outputs are annotated with contracts, which are user-defined predicates, i.e. `prime?`.

```
keygen() : prime? { • }
rsa(k: prime?, s: string?) : string? { • }
rsa(keygen(), "Plain");
```

Under our modular semantics, the program executes as follows:

```
rsa(keygen(), "Plain");
→ rsa([prime?], "Plain");
→ string?("Plain"); prime?([prime?]); [string?]
→ [string?]
```

The `[·]` notation denotes a contract treated as a value. Intuitively, it represents the set of all values satisfying the contract. The implementation of `keygen` is missing, so we cannot know what it returns, but by its specification, it produces a value satisfying `prime?`, hence it produces `[prime?]`. To call `rsa`, we check `string?` of "Plain" and `prime?` of `[prime?]`, both of which succeed, so the program produces `[string?]`, an unknown string value. No contracts are violated and thus expensive run-time checks can be eliminated.

To obtain an analysis, this *modular* reduction semantics is run through the same derivation pipeline to reveal a *modular* program analysis. The resulting analysis is easy to verify, extend, and implement, requiring no ingenuity in approximation methods.

The central lesson of this work is that *problems in program analysis can be solved by developing novel program semantics and deriving abstractions conventionally*. Generalizing this observation, we can see that this strategy applies to many analysis problems. Determine the question to be answered, design a semantics that precisely answers this question during evaluation, as in our modular semantics, and finally, use traditional transformations and approximation methods to produce a computable analyzer.

This strategy has several advantages: (1) It is easier to get right. Semantics and analysis correspond closely, making both easier to verify and maintain. (2) Many existing semantics can be repurposed for building analyses of everything from space behavior of lazy languages to security via stack inspection. (3) The PL community has developed a host of intellectual tools for designing and reasoning about semantics which we can re-use for program analysis.

Future Fun

We have taken this approach to leverage dynamic semantics for predicting garbage collection, space consumption, and modularity. There are many exciting opportunities we consider worth pursuing.

1. Using a parallel cost model semantics [6], we can design analyses for predicting space usage of parallel functional programs.
2. Contracts are a form of specification, which we treat as values in a novel semantics. What other kinds of specifications can be treated as values? Giving reductions for values drawn from Hoare-type theory [7] would give rich specifications for effectful components, in turn yielding rich program analyzers.
3. Using a semantics with temporal predicates over program events [8], we can develop higher-order temporal model checkers for history- and stack-based security mechanisms.

These problems seem daunting under current approaches to analysis design, but we conjecture that by taking a semantic approach seriously, solutions will be more easily obtained.

Acknowledgments: We are inspired in part by work with M. Might.

References

- [1] O. Shivers. Control flow analysis in Scheme. In *PLDI '88*, pages 164–174.
- [2] M. Might, Y. Smaragdakis, and D. Van Horn. Resolving and exploiting the *k*-CFA paradox. In *PLDI '10*, pages 305–315.
- [3] P. Meunier, R. B. Findler, and M. Felleisen. Modular set-based analysis from contracts. In *POPL '06*, pages 218–231.
- [4] S. Tobin-Hochstadt and D. Van Horn. Modular analysis via specifications as values. *CoRR*, abs/1103.1362, 2011.
- [5] D. Van Horn and M. Might. Abstracting abstract machines. In *ICFP '10*.
- [6] D. Spoonhower, G. E. Blelloch, R. Harper, and P. B. Gibbons. Space profiling for parallel functional programs. *JFP*, 20(5-6):417–461, 2010.
- [7] A. Nanevski, G. Morrisett, and L. Birkedal. Hoare type theory, polymorphism and separation. *JFP*, 18(5-6):865–911, September 2008.
- [8] C. Skalka, S. Smith, and D. Van Horn. Types and trace effects of higher order programs. *JFP*, 18(2):179–249, March 2008.