On the Relationship Between Formal Semantics and Static Analysis

P. N. Benton*, University of Cambridge

Introduction

Static analysis is the automatic compile-time analysis of programs so as to extract information which can be used to produce optimised code. Much work on static analysis draws on more theoretical research into the semantics of programming languages, as this can provide frameworks both for formulating analyses and for proving their correctness – such relationships between 'pure' and 'applied' fields are common throughout the sciences. This paper highlights a few existing interactions between the two fields and identifies some current trends and possible directions for future work. The account is (unsurprisingly) biased towards my own interests in functional computation.

It seems entirely natural that static analysis should be a major consumer of, as well as a testbed and an inspiration for, ideas in semantics because both fields are, at their broadest, concerned with finding formal principles for reasoning about the behaviour of programs. The major distinction is that static analyses are intended to be implemented, and must therefore be (efficiently) computable. Since nearly all interesting program properties are non-computable, static analyses cannot discover precise information and we must instead accept safe, and often rather crude, approximations. Semanticists, by contrast, are generally interested in modelling programming languages as accurately as possible and in proving rather more complex program properties than are considered in static analysis. Thus the essential difference is in the level of abstraction at which one works, though the important question of how analyses should be implemented has no obvious counterpart in semantics.

Analysis techniques

The idea of the previous paragraph is, roughly, the basis for abstract interpretation [3]. An abstract interpretation assigns, usually compositionally, an abstract meaning in some lattice of properties to each program phrase. Correctness is established by relating this abstract semantics to a more standard concrete semantics.

The other main way of presenting analyses is to give a logic or type system in which one can derive judgments asserting that a program phrase satisfies a particular predicate. The relationship between abstract interpretations and inference-based analyses, in terms of both theory and implementation, is an active research area in which non-trivial semantic ideas, such as Stone duality, have been influential [1], and there is scope for more work along similar lines, particularly on the analysis of concurrent systems. In a sense which is so broad as to be useless, it is probably correct to say that the two frameworks are

*Supported by EU BRA 8130 LOMAPS.

'equivalent'; but the qualitative differences are still significant, and what is natural in one framework may be awkward in the other. One sociological benefit of using analysis logics is that the connections with more general and well-known program logics and type systems become obvious, and ordinary type inference can be seen as a static analysis problem. Abstract interpretation is sometimes unfairly seen as a rather specialised subject.

Semantics in static analysis

Domain theory has been used to formulate and validate many static analyses, particularly for functional languages [2, 1, 4]. Analyses of properties such as strictness, binding-time or totality are often not merely validated by such a semantics, but are formulated right from the start in semantic terms; domain-theoretic concepts, such as projections, can naturally suggest program analysis techniques [4]. More sophisticated semantic concepts, such as parametricity [1], initiality, fibrations [4] and higher-dimensional automata, have also been used in static analysis. Operational, rather than denotational, semantics are now often used as the basis for static analyses: see, for example, [6, 8].

Although static analysis sometimes uses moderately sophisticated ideas from the semantics community, this is comparatively rare – more often static analysis develops its own theory on top of fairly elementary semantics. There are several possible explanations for this:

1. Relevant semantic work is not well-known or its relevance is not widely appreciated. For example, the monadic approach to semantics [5] is well-known to theorists, but still has unrealised potential as a framework for understanding 'type and effect' analyses like [8].

2. Relevant semantic work does not exist, because the properties of interest are at a lower level or are more intensional than is usually dealt with in semantics (e.g. data representation, memory allocation), or because semanticists have not yet dealt satisfactorily with certain high-level language features. A lack of denotational models for concurrent languages was one reason for the move to operational techniques in static analysis, though such models are now starting to appear.

3. "Semanticists only deal with toy languages". (cf. 2) Although this contains a grain of truth, theory provides paradigms, insights and intuitions which are more accessible and more applicable because they are not tied up with the details of particular languages. Turning theory into practice is the job of practitioners, though for this to happen theoreticians must make their work relevant and accessible.

4. "Many of the subtleties studied by semanticists can safely be ignored because static analysis only deals with approximations." For example, although simple domain-theoretic semantics are often not fully abstract, they are useful in static analysis. However, for some applications, the approximations introduced by a naive semantics may be particularly bad ones. Also, the payoff from semantic work may be more indirect, as is shown by the use of intensional representations of functions, developed in the search for fully abstract models of PCF, to write efficient abstract interpreters.

5. "Semanticists study notions of program *equivalence* and look for models which reflect these equivalences as accurately as possible. Static analysis is only concerned with program *properties*, which can be interpreted in a naive semantics." (cf. 4) One

reason that program equivalence is rarely mentioned in static analysis is that the correctness of the associated optimising transformations is only rarely addressed. The purpose of analysis is to replace one program with another, and a notion of equivalence is needed to justify this. Furthermore, even if the properties have naive interpretations, the entailment relation between those properties is refined by moving to a more accurate semantics, enabling more useful information to be gathered without necessarily increasing the complexity of the analysis.

The question of how to justify optimising transformations is clearly important and may provide a bridge between static analysis and more theoretical work concerning languages with refined type systems, which *enforce* certain behavioral constraints. This work is essentially dual to static analysis and often has strong semantic or proof-theoretic foundations. Analysis-based optimisations may be expressed via translations into such a refined language, and it seems natural to design the analysis after the type system of the target language, giving a primary role to logical or proof-theoretic ideas.

Situations 1 and 2 above can be improved by more collaboration. A good example of where this might be profitable is current research into the semantics of local variables [7]; this seems closely related to the problems of scope and aliasing which arise in static analysis of dynamic creation of storage regions [8] and communication channels [6]. This is also an example of the situation mentioned in 4 and 5: many of the 'tricky' program equivalences arising from local names and higher-order functions correspond to plausible optimisations, yet they are only validated by fairly complex semantics. Analyses and semantics should develop together.

References

[1] P. N. Benton. Strictness logic and polymorphic invariance. In *Proceedings of the Second Symposium on Logical Foundations of Computer Science, Tver*. Springer-Verlag LNCS 620. July 1992.

[2] G. L. Burn, C. L. Hankin and S. Abramsky. *The theory and practice of strictness analysis for higher-order functions*. Science of Computer Programming 7. 1986.

[3] P. Cousot and R. Cousot. *Abstract Interpretation Frameworks*. Journal of Logic and Computation 4(2) 1992.

[4] J. Launchbury. *Projection factorisations in partial evaluation*. Distinguished Dissertations in Computer Science Vol. 1. Cambridge University Press 1991.

[5] E. Moggi. *Notions of Computation and Monads*. Information and Computation 93(1) 1991.

[6] F. Nielson and H. R. Nielson. *From CML to its Process Algebra*. To appear in Theoretical Computer Science, North-Holland.

[7] I. D. B. Stark, *Categorical Models for Local Names*. Lisp and Symbolic Computation 9(1) 1996.

[8] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. *Proceedings of the 21st ACM Symposium on Principles of Programming Languages*. ACM Press. January 1994.