Ott

Stephanie Weirich

Ott evangelist

University of Pennsylvania

Abstract

This talk discusses the experience of using Ott for programming language design.

Categories and Subject Descriptors D.3.1 [**Programming Languages**]: Formal Definitions and Theory

General Terms Languages, Theory, Verification, Standardization

I plan to use Ott in every new paper that I write, in some form. The tool has become an important part of my design process, and I have come to rely on it. The purpose of this (part of) the talk is to explain why.

Ott is a tool for specifying the concrete and abstract syntax of programming languages and systems of inference rules that specify the semantics. From this specification, Ott can generate definitions in LaTeX for typesetting, OCaml for implementation, and Coq, Isabelle/HOL or HOL4 for formal mathematics. The input language to Ott is concise and resembles an email that you might send to your coauthors.

However, this talk is not about the mechanical formalization of programming language *metatheory*. Ott provides a range of uses and, although I have used Coq to prove properties about language specifications generated by Ott, this is not my main mode of use. Instead, the majority of the benefit that I get from Ott is the mechanical formalization of programming language *specifications*.

By specifying the semantics of a programming language (or a simple toy calculi) in an Ott file, then language design becomes a tool-assisted activity instead of pure mathematics. The Ott file can be part of a version repository, so several (geographically distributed) coauthors can work on the design simultaneously, using the most up-to-date definitions. The LaTeX output means that not all coauthors need to understand the Ott input language. Rules are organized and consistently named, so the language specification is concentrated in the Ott files, not scattered and duplicated across a number of tex files.

The process of specifying a language using Ott provides a lightweight form of consistency checking. Definitions in the semantics must parse, ruling out typos and unintentional ambiguity. Notations and metaproductions give flexibility to the specification, while still leaving traces in the Ott input so they cannot be completely informal. Further consistency checking comes from proof assistant code generation—then not only must the definitions parse, they also must typecheck. These consistency checks aid collaboration as much as the final presentation of the material for publication.

The primary advantage that Ott gives is flexibility in the design process. With this flexibility, I can search a much larger space of potential designs more effectively. Part of this flexibility is due to flexible grammars: Syntactic changes are often one line changes to the Ott file. (And, I hate to admit it, but changing the syntax of an object language can often lead new insight into its design.)

However, part of the flexibility is due to the consistency checks. Just as typed languages (such as ML and Haskell) are easier to refactor because the type checker helps to identify all of the places in the source code that changes are needed, Ott can identify all of the ramifications of specificational changes. This makes it difficult

Or Nott

Scott Owens **Peter Sewell** **Francesco Zappa Nardelli**

Ott developers

University of Cambridge INRIA

Abstract

We reflect on the limitations of Ott, and on what other (*New Ott*?) tool support a working semanticist might want in an ideal world.

Categories and Subject Descriptors D.3.1 [**Programming Languages**]: Formal Definitions and Theory

General Terms Languages, Theory, Verification, Standardization

We developed Ott[a] to provide tool support 'for the working semanticist', and it has been used, by ourselves and others, in a fair number of substantial projects. However, there are still many challenges for the future: areas where existing tools (including Ott) are lacking. This (part of the) talk raises a few of them.

Parsing and Pretty-Printing Ott takes a user specification of an arbitrary context-free grammar (with subgrammars and list forms) and builds a parser, to use for parsing semantic rules and examples. The flexibility that this affords, letting the user freely define whatever (potentially ambiguous) object language and formula grammar they need, and without heavy quoting and antiquoting to move between them, is very useful. However, Ott does not build a standalone and production-quality parser that could be used in a full-scale language implementation; nor does it build a standalone pretty-printer for abstract syntax terms.

Semantics without Syntax Ott shines in cases where the semantics of the object language is expressed principally in terms of a free syntax, e.g. for structured operational semantics and type systems. Outside that domain, e.g. when one deals with the sequential semantics of machine code (with little syntax but much bit manipulation) or with axiomatic relaxed-memory concurrency semantics (expressed with first-order axioms about relations over events), it gives little or no benefit. Instead, one needs good libraries for finite sets, lists, and so on.

The Ott Type System Considered as a type system, Ott grammars can make use of mutually recursive labelled sums-of-products, with subtyping arising from subgrammar declarations (e.g. for a *value* subgrammar of some *expressions*). This serves surprisingly well, but when one wants to start defining functions one quickly also wants top-level parametric polymorphism and perhaps also type classes.

Binding One of the starting points for the Ott development (which began in late 2004), was the realisation that dealing with rich forms of binding becomes important when one goes beyond small calculi; it introduced a broad class of binding specifications. Implementing that (up to alpha conversion) in full generality remains a challenge, and is perhaps too much to aim for — but Ott can now generate the Locally Nameless representation in relatively simple cases (with further proof infrastructure provided by Aydemir and Weirich's LNgen tool). The Nominal Isabelle system now has direct support for a moderately large subset of Ott-like binding specifications.

However, while dealing with binding is certainly essential for some applications, we find many in which it is not an important

[a] Sewell, Zappa Nardelli, Owens, Peskine, Ridge, Sarkar, and Strniša, JFP 20(1), 2010; Invited submission from ICFP 2007

to miss unintended consequences of such changes. As the system evolves, I do not reprove all of the properties that I think it should have, but I do appreciate the opportunity to reexamine all of the parts of the specification that might invalidate those properties.

Certainly, this process does not provide as much confidence in the correctness of the design as mechanical proofs of metatheory, but it requires much less effort and can be extended to a mechanical proof at a later date. Although the LaTeX output may not be as beautiful (or concise) as in a hand-crafted paper, the real benefits for collaboration and exploration are worth the trouble, and in the end, lead to better designs.

issue. For example, in our $\text{OCaml}_{\text{light}}$ semantics we could use the fully concrete representation except for a very modest De Bruijn encoding for type variable binders, and in current work on processor semantics there is no binding whatsoever.

Executable Semantics The last part of the POPLmark challenge focussed on making a semantics executable in some form. We would like to re-emphasise its importance: in our view, two primary uses of a semantic definition should be (a) exploring its consequences on examples, at design-time, and (b) testing conformance between it and an implementation (until the day when full compiler verification becomes routine).

Tighter Prover Integration Ott can be used as a stand-alone tool (doing some checking and producing LaTeX) or as a front-end to a prover. In principle, though, many of the ideas could and should be integrated into prover user interfaces, preferably abstracting from the details of the individual provers as much as possible. This would obviously be a big engineering challenge.

Conclusion In shared conclusion, we would like to note that while mechanising proofs is highly worthwhile, mechanising definitions is even more important, and is a substantial challenge in itself.