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 meta-theory. Ott provides a range of uses and, although my coauthors and 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 calculus) 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 specification changes. This makes it difficult
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.

**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.