Verifying type soundness in HOL for OCaml: the core language

Scott Owens Gilles Peskine

University of Cambridge

Despite the recent interest in using computerized theorem provers to verify theorems in programming languages research, little progress has been made toward verification for real-world languages. Such undertakings encounter two difficulties: not only are the proofs larger and more complicated than in a corresponding research calculus, but the language itself typically lacks a formal specification. We attack both of these problems for OCaml, formalizing a sizable fragment of the language using Ott [5] and verifying its type soundness in HOL [4].

We take a straightforward approach and formalize the syntax of OCaml as a collection of (sometimes mutually) inductive datatypes, and OCaml's type system and operational semantics as a collection of (sometimes mutually) inductive relations and recursive functions. Nipkow and van Oheimb[3] apply a similar methodology to Java. Since OCaml does not come with a formal specification of its type system or operational semantics, we base our semantics on the informal language manual, and on the results of experiments with the implementation. Ott allows us to express these definitions in a mathematical-style notation. It generates HOL, Coq, Isabelle/HOL, and LaTeX definitions, the first of which forms the basis for our proof.

In contrast to Lee et al. [2], we specify the semantics directly on the language's source syntax, rather than using an elaboration into an internal calculus. The direct approach requires reasoning about diverse and redundant features in the verification, but the theorem prover helps us manage the process. We hypothesize that a clear connection between the source language and its semantics will facilitate verification about OCaml programs in addition the present verification about the OCaml language.

Formalization Our semantics covers a large part of OCaml, but excludes the language's module and object systems. We formalize:

- let-based polymorphism with a traditional value restriction
- pattern matching, with nested and "or" patterns; without guards
- type definitions, including type abbreviations, generative variant and record types, parametric types, and mutual recursion
- exception definitions and handling (`try`, `raise`, `exception`)
- mutable references (`ref`, `!`, `:=`), but not mutable record fields
- 31 bit word semantics for values of type `int`
- polymorphic equality (`=`), tuples (`,`), and lists (`::`, `[]`)
- record update expressions (`with`) and loops (`while`, `do`).

The syntax specification closely follows the OCaml manual, with several additions to support the semantics: typing contexts, stores, type schemes, names of primitive functions, etc. The type system has 156 rules in 32 relations, not including substitution and free variable functions generated by Ott. The typing relation is syntactic, but not unification-based, so special care is taken to coordinate type variables mentioned in explicit type annotations.

The small-step reduction relation uses labeled transitions to express interactions with the store. The rules for the pure fragment thereby avoid mentioning the store, similar to the "state convention" of *The Definition of Standard ML*, but with formal rigor. This approach also simplifies the introduction of new effectful features by localizing their mention to the new rules. There are 18 rules for evaluating in contexts (they enforce right-to-left evaluation ordering), 17 for upward propagation of exceptions, and 26 for reducing expressions, 13 for primitive equality, 11 for other primitive operations, and 12 for matching values against patterns.

Verification Most of the proof effort is a straightforward application of HOL's tactic-based proof mechanisms. Ott uses HOL's built-in facilities for datatype and inductive relation definitions, and these automatically provide rule- and structural-induction theorems, and case analysis theorems. Additionally, we have automated the construction of inversion lemmas from the provided case analysis theorems. We use HOL's tactics for the following operations: rewriting with equational theorems, backward and forward chaining with implicational theorems, instantiating existentially quantified variables, applying induction theorems, case-splitting, and doing first-order proof search (`METIS_TAC` [1]). The last is used on around 500 goals in the verification; a good prover for first-order logic with equality is an immense convenience in large verifications.

The proof does not require an α-aware term or type representation, but we have settled on a de Bruijn index encoding of type variables. Because the semantics never performs a reduction under a value or type variable binder, identifying names up to α-equivalence is unnecessary; the Ott-generated non-capture-avoiding substitutions will never capture because the term being substituted has no free variables. Type variables, bound by polymorphic `let`, have the additional constraint that duplicate bindings cannot be present in the typing context. Otherwise type scheme generalization might capture a type variable introduced by a different `let` expression. Unfortunately, in the polymorphic `let` case of the weakening lemma's proof, a new type variable might conflict with one in the weakened context. Thus, the `let` expression's typing derivation must be α-renamed to introduce a non-conflicting name. Preliminary investigation of two α-unaware approaches to this dilemma showed the resulting proof obligations to be unsatisfactorily complicated. The de Bruijn index encoding sidesteps the potential for naming conflicts. Typical "locally nameless" encodings do not help here because the binding in question spans two positions in the judgment.

Status We have completed the verification of the progress theorem for expressions (without type abbreviations). Except for a combination weakening/type substitution lemma, we have also completed verification of the preservation theorem for expressions using the named representation for type schemes. The total proof effort uses about 4,400 lines of tactic scripts and has taken roughly 2.5 man-months thus far (the formalization has taken an estimated .5 to 1 man-month). Ongoing work aims to replace the named representation with a de Bruijn index representation to simplify the verification of the (now separate) weakening and type substitution lemmas, and to extend the proof to handle type abbreviations. The formalization and proofs are available at `http://www.cl.cam.ac.uk/~pes20/ott/`.

Acknowledgments We acknowledge the support of EPSRC grants GR/T11715/01 and EP/C510712/1.

References

[1] HURD, J. First-order proof tactics in higher-order logic theorem provers. In *Proc. Workshop on Design and Application of Strategies/Tactics in Higher Order Logics* (2003).

[2] LEE, D. K., CRARY, K., AND HARPER, R. Towards a mechanized metatheory of Standard ML. In *Proc. Principles of Prog. Lang.* (2007).

[3] NIPKOW, T., AND VAN OHEIMB, D. Java$_{light}$ is type-safe — definitely. In *Proc. Principles of Programming Languages* (1998).

[4] NORRISH, M., AND SLIND, K. *HOL-4 manuals*, 1998–2006. `http://hol.sourceforge.net/`.

[5] SEWELL, P., ZAPPA NARDELLI, F., OWENS, S., PESKINE, G., RIDGE, T., SARKAR, S., AND STRNIŠA, R. Ott: Effective tool support for the working semanticist. In *Proc. International Conference on Functional Programming* (2007).