Lem: A Lightweight Tool for Heavyweight Semantics

Scott Owens[1], Peter Böhm[1], Francesco Zappa Nardelli[2], and Peter Sewell[1]

[1] University of Cambridge
[2] INRIA
www.cl.cam.ac.uk/users/so294/lem

Abstract. Many ITP developments exist in the context of a single prover, and are dominated by proof effort. In contrast, when applying rigorous semantic techniques to realistic computer systems, engineering the definitions becomes a major activity in its own right. Proof is then only one task among many: testing, simulation, communication, community review, etc. Moreover, the effort invested in establishing such definitions should be re-usable and, where possible, irrespective of the local proof-assistant culture. For example, in recent work on processor and programming language concurrency (x86, Power, ARM, C++0x, CompCertTSO), we have used Coq, HOL4, Isabelle/HOL, and Ott—often using multiple provers simultaneously, to exploit existing definitions or local expertise.

In this paper we describe Lem, a prototype system specifically designed to support pragmatic engineering of such definitions. It has a carefully designed source language, of a familiar higher-order logic with datatype definitions, inductively defined relations, and so on. This is typechecked and translated to a variety of programming languages and proof assistants, preserving the original source structure (layout, comments, etc.) so that the result is readable and usable. We have already found this invaluable in our work on Power, ARM and C++0x concurrency.

1 Motivation

Mechanised proof assistants such as ACL2 [1], Coq [6], HOL4 [9], HOL Light [8], Isabelle/HOL [10], PVS [12], and Twelf [19] are becoming important tools for Computer Science. In many applications of these tools, the majority of effort is devoted to proof, and that is rightly a main focus of their developers. This focus leads each of these systems to have its own logic, various mechanisms for making mathematical definitions, and extensive support for machine-checked interactive and/or automated reasoning.

In some applications, however, the *definitions* themselves, of types, functions, and relations, are a major focus of the work. This is often the case when modelling key computational infrastructure: network protocols, programming languages, multiprocessors, and so on. For example, we have worked on TCP [4,13], Optical Networking [3], Java Module Systems [18], the semantics of an OCaml fragment [11], concurrency for C and C++ [2,5,16], and the semantics of x86, POWER and ARM multiprocessors [14,15]; and there are numerous examples by other groups (far too many to cite here). In each of these cases, considerable

effort was required to establish the definitions of syntax and semantics, including analysis of informal specifications, empirical testing, and proof of metatheory. These definitions can be large: for example, our TCP specification is around 10 000 non-comment lines of HOL4. At this scale, the activity of working with the definitions becomes more like developing software than defining small calculi: one has to refactor, test, coordinate between multiple people, and so on, and all of this should, as far as possible, be complete before one embarks on any proof.

Moreover, in such work a proof assistant is just one piece of a complex project, involving production typesetting, testing infrastructure, code generation, and tools for embedding source-language terms into the prover. Sometimes there is no proof activity, but great benefits arise simply from working in typechecked and typeset mathematics; sometimes there is mechanised symbolic evaluation or code generation for testing and prototyping; sometimes there is hand proof or a mixture of hand and mechanised proof; and sometimes there is the classic full mechanised proof supported by provers.

Ideally, the results of such work should be made widely available in a *re-usable* form, so that other groups can build on them and so that the field can eventually converge on standard models for the relatively stable aspects of the computational environment in which we work. Unfortunately, at present such re-use is highly restricted for two reasons. Firstly, the field is partitioned into schools around each prover: the difficulty in becoming fluent in their use means that very few people can use more than one tool effectively. Indeed, even within some of our own projects we have had to use several provers due to differing local expertise. This variation makes it hard to compare the results of even carefully specified benchmarks, such as the POPLmark challenge.

Secondly, the differences between the provers mean that it is a major and error-prone task to port a development—or even just its definitions—from one system to another. In some cases this is for fundamental reasons: definitions which make essential use of the dependent types of Coq may be hard or impossible to practically port to HOL4. However, many of the examples cited above are logically undemanding: they have no need for dependent types, the differences between classical and constructive reasoning are not particularly relevant, and there is often little or no object-language variable binding (of course this does not apply for formalisation of rich type theories). They do make heavy use of basic discrete mathematics and "programming language" features: sets and set comprehensions; first-order logic; and inductive types and records with functions and relations over them. Thus, the challenge is one of robustly translating between the concrete syntax and definition styles of the different proof assistants.

2 Portable definitions with Lem

We have designed a language, Lem, for writing, managing and publishing large scale semantic definitions, for use as an intermediate language when generating definitions from domain-specific tools, and for use as an intermediate language for porting definitions between existing provers. Our implementation can currently typecheck Lem sources, and generate HOL4, Isabelle/HOL, OCaml, and

LaTeX (the latter drawing on Wansborough's HOLDoc tool design). Development of a Coq backend is in progress. We are already using Lem in our research: we developed a semantics for multiprocessor concurrency on the POWER architecture [14] in Lem, and our semantics for C++0x [2,5] concurrency has been ported from Isabelle/HOL to Lem.

Semantically, we have designed Lem to be roughly the intersection of common functional programming languages and higher-order logics, as we regard this as a sweet spot: expressive enough for the applications we mention above, yet familiar and relatively easy to translate into the various provers; there is intentionally no logical novelty here. Lem has a simple type theory with primitive support for recursive and higher-order functions, inductive relations, n-ary tuples, algebraic datatypes, record types, type inference, and top-level polymorphism. It also includes a type class mechanism broadly similar to Isabelle's and Haskell's (without constructor classes). It differs from the internal logics of HOL4 or Isabelle/HOL principally in having type, function and relation definitions as part of the language rather than encoded into it: the Lem type system is formally defined (using Ott [17]) in terms of the user-level syntax.

The novelty is rather in the detailed design and implementation, to ensure the following four important pragmatic properties. We can achieve all of these goals more easily than one could in context of a prover implementation because we are not constrained to use an intermediate representation suitable for the implementation of a proof kernel (e.g., explicitly typed lambda terms), and because we are building a lightweight stand-alone tool, without a large legacy codebase.

1. Readability of source files Lem syntactically resembles OCaml and F#, giving us a popular and readable syntax. It includes nested modules (but not functors), recursive type and function definitions, record types, type abbreviations, and pattern matching. It has additional syntax for quantifiers, including restricted quantifiers ($\forall x \in S.\ Px$), set comprehension, and inductive relations. For example, here is an extract from our POWER model:

```
let write_reaching_coherence_point_action m s w =
  let writes_past_coherence_point' =
    s.writes_past_coherence_point union {w} in
  (* make write before other writes to this address not past coherence *)
  let coherence' = s.coherence union
      { (w,wother) | forall (wother IN (writes_not_past_coherence s))
      | (not (wother = w)) && (wother.w_addr = w.w_addr)}  in
  <| s with coherence = coherence';
            writes_past_coherence_point = writes_past_coherence_point' |>
let sem_of_instruction i ist =
  match i with
  | Padd  set rD rA rB  -> op3regs    Add set rD rA rB ist
  | Psub  set rD rA rB  -> op3regs    Sub set rD rB rA ist (* swap args *)
  end
```

We do not always follow OCaml: for example, Lem uses curried data constructors instead of tupled ones, and it uses `<|` and `|>` for records, saving `{` and `}` for set comprehensions. Type classes provide principled support for overloading.

Lem does not at present include support for arbitrary user-defined syntax, as provided by Ott [17] and (to a greater or lesser extent) by several proof assistants. Lem and Ott have complementary strengths: Ott is particularly useful for defining semantics as inductively defined relations over a rich user syntax, but has limited support for logic, sets, and function definitions, whereas Lem is the converse. We envisage refactoring the Ott implementation, which currently generates Coq, HOL, and Isabelle/HOL code separately, to instead generate Lem code and leave the prover-specific output to the Lem tool. In the longer term, a metalanguage that combines both is highly desirable.

2. Taking the source text seriously *Explaining* the definitions is a key aspect of the kind of work we mention above. We need to produce production-quality typesetting, of the complete definitions in logical order and of various excerpts, in papers, longer documents, and presentations. As all these have to be maintained as the definitions evolve, the process must be automated, without relying on cut-and-paste or hand-editing of generated LaTeX code. Moreover, it is essential to give the user control of layout. Here again the issues of large-scale definitions force our design: in some cases, especially for small definitions, pretty printing from a prover internal representation can do a good enough job, but manual formatting choices were necessary to make (e.g.) our C++0x memory model readable. Accordingly, we preserve all source-file formatting, including line breaks, indentation, comments, and parentheses, in the generated code. This lets us generate corresponding LaTeX code, e.g. for the previous example:

```
let write_reaching_coherence_point_action m s w =
  let writes_past_coherence_point' =
    s.writes_past_coherence_point ∪ {w} in
  (* make write before other writes to this address not past coherence *)
  let coherence' = s.coherence ∪
      {(w, wother)|∀wother∈(writes_not_past_coherence s)
      | (¬ (wother = w)) ∧ (wother.w_addr = w.w_addr)} in
  ⦇s with coherence = coherence';
          writes_past_coherence_point = writes_past_coherence_point'⦈

let sem_of_instruction i ist =
  match i with
  | Padd set rD rA rB → op3regs Add set rD rA rB ist
  | Psub set rD rA rB → op3regs Sub set rD rB rA ist (* swap args *)
  end
```

It also ensures that the generated prover and OCaml code is human-readable in its own right.

3. Support for execution *Exploring* such definitions, and *testing* conformance between specifications and deployed implementations (and between specifications at different levels of abstraction), is also a central aspect of our work; both need some way to make the definitions executable. In previous work with various colleagues we have built hand-crafted symbolic evaluators within HOL4 [3,4,13,15], interpreters from code extracted from Coq [16], and memory model exploration tools from code generated from Isabelle/HOL [2]. Lem

supports several constructs which cannot in general be executed, e.g., quantification in propositions and set comprehensions, but Lem can generate OCaml code where the range is restricted to a finite set (otherwise OCaml generation fails). This has been invaluable for our POWER memory model exploration tool [14].

4. Quick parsing and type checking with good error messages This is primarily a matter of careful engineering, using conventional programming-language techniques. Lem is a batch-mode tool in the style of standard compilers, rather than focussed on interactive use, in the typical proof-assistant style.

3 Implementation

Our Lem implementation is written in OCaml, using Ott to specify the concrete syntax, and it loosely follows the architecture of a traditional compiler. The central data structure is a typed abstract syntax tree (AST), and processing follows 4 phases: (1) source files are lexed and parsed into untyped ASTs; (2) the untyped ASTs are type checked and converted into typed ASTs; (3) typed-AST-to-typed-AST transformations remove language features that are not present in the target (e.g., the removal of type classes by introducing dictionary passing for OCaml and HOL4); and (4) the transformed, typed AST is printed in the target language syntax. We try to make the printing step as simple as possible, and uniform across the various back-ends, by handling all of the complexities of translation in (3). The untyped and typed ASTs contain all of the whitespace (both indentation and line breaks) and comments of the original source file; the step (4) printer uses these instead of a pretty printing algorithm for layout.

The logical design of Lem makes the basic translation to a variety of targets straightforward. The standard libraries of our various targets have differing data representations and interfaces. For each desired feature (e.g., finite maps, or bit vectors), we design an interface for Lem, and specify how that interface is to be translated for each target. This is similar to Ott's *hom* functionality; however, here we typecheck the translation specifications to ensure that the generated code is well-formed.

4 Future work

We are actively developing Lem: our immediate goal is to finish and polish the existing backends (including the in-progress Coq backend). Also of interest is a HOL4-to-Lem translation (allowing us to automatically port, for example, Fox's detailed ARM instruction semantics [7] to other provers) and we would like Coq-to-Lem and Isabelle/HOL-to-Lem translations, which will need expertise in the front-end implementations of those systems. Lem does not currently support OCaml generation for inductively defined relations (although one can sometimes use the Isabelle backend and then apply its code generation mechanism). Ultimately, we would like to directly generate OCaml that searches for derivations; this will be particularly useful in conjunction with Ott, for running test and example programs directly on an operational semantics.

Although Lem is primarily a design and engineering project, it would benefit from a rigorous understanding of exactly how the semantics of the source and

target logics relate to each other, for the fragments we consider. In particular, when multiple provers are used to verify properties of a Lem-specified system, we would like a semantic justification that the resulting definitions have the same meaning, and that a lemma verified in one prover can be used in another. There have been several projects that port low-level proofs between provers (a very different problem to the readable-source-file porting that we consider here); while this approach yields the right guarantees, we expect it would be very challenging because the various backends can transform the same definition differently (e.g., keeping type classes for Isabelle, but not for HOL4).

References

1. ACL2 Version 4.2 (2011), `http://www.cs.utexas.edu/~moore/acl2/`
2. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. In: POPL 2011. pp. 55–66. ACM (2011)
3. Biltcliffe, A., Dales, M., Jansen, S., Ridge, T., Sewell, P.: Rigorous protocol design in practice: An optical packet-switch MAC in HOL. In: ICNP 2006. pp. 117–126. IEEE (2006)
4. Bishop, S., Fairbairn, M., Norrish, M., Sewell, P., Smith, M., Wansbrough, K.: Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and Sockets. In: SIGCOMM 2005. pp. 265–276. ACM (2005)
5. Blanchette, J.C., Weber, T., Batty, M., Owens, S., Sarkar, S.: Nitpicking C++ concurrency. In: PPDP 2011. ACM (2011), to appear
6. The Coq proof assistant, v.8.3 (2011), `http://coq.inria.fr/`
7. Fox, A.C.J., Myreen, M.O.: A trustworthy monadic formalization of the ARMv7 instruction set architecture. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 243–258. Springer (2010)
8. Harrison, J.: HOL Light (2011), `http://www.cl.cam.ac.uk/~jrh13/hol-light/`
9. The HOL 4 system, Kananaskis-6 release (2011), `http://hol.sourceforge.net/`
10. Isabelle 2011 (2011), `http://isabelle.in.tum.de/`
11. Owens, S.: A sound semantics for OCaml$_{light}$. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 1–15. Springer (2008)
12. PVS 5.0 (2011), `http://pvs.csl.sri.com/`
13. Ridge, T., Norrish, M., Sewell, P.: A rigorous approach to networking: TCP, from implementation to protocol to service. In: Cuéllar, J., Maibaum, T.S.E., Sere, K. (eds.) FM '08. LNCS, vol. 5014, pp. 294–309. Springer (2008)
14. Sarkar, S., Sewell, P., Alglave, J., Maranget, L., Williams, D.: Understanding POWER multiprocessors. In: PLDI 2011. ACM (2011), to appear
15. Sarkar, S., Sewell, P., Zappa Nardelli, F., Owens, S., Ridge, T., Braibant, T., Myreen, M.O., Alglave, J.: The semantics of x86 multiprocessor machine code. In: POPL 2009. pp. 379–391. ACM (2009)
16. Ševčík, J., Vafeiadis, V., Zappa Nardelli, F., Jagannathan, S., Sewell, P.: Relaxed-memory concurrency and verified compilation. In: POPL 2011. pp. 43–54. ACM (2011)
17. Sewell, P., Zappa Nardelli, F., Owens, S., Peskine, G., Ridge, T., Sarkar, S., Strniša, R.: Ott: Effective tool support for the working semanticist. JFP 20(1) (Jan 2010)
18. Strniša, R., Sewell, P., Parkinson, M.: The Java Module System: core design and semantic definition. In: OOPSLA 2007. pp. 499–514. ACM (2007)
19. Twelf 1.5 (2011), `http://twelf.plparty.org/wiki/Main_Page`