Expressing Large-scale Semantics in Lem

Scott Owens, Peter Böhm, Mark Batty, and Peter Sewell

University of Cambridge
http://www.cl.cam.ac.uk/~so294/lem/

Abstract. Trustworthy software verification relies on accurate semantic definitions of the programming languages and systems that the software runs on; engineering these definitions for realistic systems is a major activity in its own right. Thus, in a large-scale verification effort, proof is only one task among many: testing, simulation, communication, community review, etc., all of which benefit from mechanization. Moreover, the effort invested in establishing such definitions should be re-usable and, where possible, not tied to a single tool.

This paper details Lem, a system specifically designed to support pragmatic engineering of such definitions. Lem's specification language is based on familiar concepts from functional programming languages and higher-order logics, and Lem translates specifications to a variety of programming languages and proof assistants, preserving the original source structure (layout, comments, etc.) to ensure the result is readable and usable. We have used Lem to support our recent work on the formalisation of C++11 concurrency. Here, we detail how Lem solved a series of common tool chain issues to give us the flexibility to use a variety of existing tools (HOL4, Isabelle, OCaml, LaTeX) in our work.

1 Motivation

The ever growing complexity of today's computer systems has established a demand for machine-checked formalization of verification efforts. Mechanised proof assistants, such as ACL2 [1], Coq [12], HOL4 [16], HOL Light [15], Isabelle/HOL [17], PVS [26], and Twelf [33] provide extensive support for machine-checked, interactive and/or automated reasoning, but their main focus is on proof support. Each has its own logic, various mechanisms for stating mathematical definitions, and different methods to formulate semantics in support of the proof.

Traditionally, the overall verification effort is dominated by proof; however, when mechanizing large-scale developments, especially those including real-world programming languages or micro-processors, good definitions are equally important. Their engineering is a major activity in its own right:

- proof is then only one task among many—testing, simulation, communication, community review, etc—which rely on the definitions; and
- concise, well-engineered definitions can support and reduce the verification effort significantly.

Additionally, in some applications 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 [7,27], Optical Networking [6], Java Module Systems [32], the semantics of an OCaml fragment [23], concurrency for C and C++ [5,9,30], and the semantics of x86, POWER and ARM multiprocessors [28,29]. The L4-verified project defines a micro-kernel, at varying levels of abstraction, in C, Haskell, and Isabelle/HOL [19]. The Verisoft project defines a micro-kernel in a subset of C (C0) with inline assembler; the formal semantics of the kernel [2] and of a C0 comiler [21] are defined in Isabelle/HOL, exclusively.

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 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 [3].

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.

1.1 Formalising C++ Concurrency

In this paper, we detail our use of Lem in formalising the concurrency model of C++ [5]. Modern multiprocessors provide shared memory as a communication channel between their individual cores, but each core implements optimisations which can visibly affect the shared memory communication, such as buffering, caching, and speculation. Compiler optimisations can further complicate the observable behaviour. To specify the observable effects of the possible reorderings, processors and progamming languages provide a relaxed interface to memory: there is no longer a notion of a global ordering of memory reads and writes. A set of rules, the *memory model*, defines the valid executions of a program by a processor. Upcoming versions of the C++ and C languages (C++11 and C1x respectively) implement a relaxed memory model that allows efficient implementation of the concurrency features running on modern relaxed memory multiprocessors.

In practice, the rules that make up the C++11 memory model—which C1x shares—are written in 'Standardese': English prose, with a subset of words like 'should' and 'shall' taking precise new meanings. The rules are numerous, intricate, and therefore prone to be incorrect or inconsistent. This combination of an informal language and an intrinsically complex specification problem makes such a document an exemplary application area for formal sematics: by understanding and specifying the document in machine-processed, formal mathematics, (unintentional) ambiguities are clarified and often first flaws are already detected before even embarking on any verification effort. Then, the semantic model can be the basis for further verification challenges, such as software or compiler verification.

2 Portable Definitions with Lem

We have designed Lem as a stand-alone, lightweight tool for writing, managing, and publishing large-scale semantic definitions, for use as a intermediate language when generating definitions for domain-specific tools, and for use as a specification language for porting definitions between existing provers. In recent work [24], we outlined a prototype version of the tool.

The tool typechecks Lem sources and generates prover code for HOL4 [16] and Isabelle/HOL [17], executable definitions in OCaml [22], and LaTeX sources; the latter drawing on Wansbrough's HOLDoc tool design. Additionally, Coq code generation is currently in development and the tool design supports the addition of new code targets in a structured, modular way.

Semantically, Lem's source language is roughly the intersection of common functional programming languages and higher-order logics, which we regard 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. The language has a simple type theory [10] 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 [25] (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 [31] in terms of the user-level syntax.

The novelty is rather in the detailed design and implementation, which ensure the following four important pragmatic properties. By building a lightweight stand-alone tool, these goals can be achieved more easily than in the context of a prover implementation because we are neither constrained by a representation suitable for a proof kernel implementation (e.g., explicitly typed lambda terms), nor by a large legacy code base.

1. Readability of source files Lem syntactically resembles OCaml and F# [14], 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 C++11 concurrency model:

```
let consistent_modification_order actions lk sb sc mo hb =
  (forall (a IN actions) (b IN actions). (a,b) IN mo
    --> (same_location a b && is_write a && is_write b)) &&
  (forall (l IN locations_of actions).
    match lk l with
      Atomic -> (
        let actions_at_l =
          {a | forall (a IN actions) | location_of a = Some l} in
        let writes_at_l =
          {a | forall (a IN actions_at_l) | is_write a} in
        strict_total_order_over writes_at_l
          (restrict_relation_set mo actions_at_l) &&
        (* hb is a subset of mo at l *)
        restrict_relation_set hb writes_at_l subset mo &&
        (* SC fences impose mo *)
        (restrict_relation_set
          (compose (compose sb (restrict_relation_set sc
            {a | forall (a IN actions) | is_fence a})) sb )
          writes_at_l) subset mo)
    | _   -> (
        let actions_at_l =
```

```
        {a | forall (a IN actions) | location_of a = Some l} in
      Set.is_empty (restrict_relation_set mo actions_at_l) )
  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 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, HOL4, 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++11 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 consistent_modification_order actions lk sb sc mo hb =
 (∀a∈actions b∈actions. ((a, b) ∈ mo)
  –> (same_location a b ∧ (is_write a ∧ is_write b))) ∧
 (∀l∈locations_of actions.
  match lk l with
   Atomic → (
    let actions_at_l =
     {a|∀a∈actions | location_of a = Some l} in
    let writes_at_l =
     {a|∀a∈actions_at_l | is_write a} in
    strict_total_order_over writes_at_l
     (restrict_relation_set mo actions_at_l) ∧
    (* hb is a subset of mo at l *)
    (restrict_relation_set hb writes_at_l subset mo ∧
    (* SC fences impose mo *)
    ((restrict_relation_set
     (compose (compose sb (restrict_relation_set sc
      {a|∀a∈actions | is_fence a})) sb)
```

```
      writes_at_l) subset mo)))
  | _ → (
      let actions_at_l =
        {a|∀a∈actions | location_of a = Some l} in
      Set.is_empty (restrict_relation_set mo actions_at_l))
  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 [6,7,27,29,8], interpreters from code extracted from Coq [30], and memory model exploration tools from code generated from Isabelle/HOL [5]. 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 [28] and has replaced our use of the Isabelle/HOL code generator for C++11.

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 C++11 Concurrency in Lem

We have used Lem to formalise the semantics of the C++11 concurrency features and its relaxed memory model. As briefly mentioned in Section 1.1, such a formalisation effort has to translate a specification written in prose English into a concise model written in rigorous mathematics, which is then used to formally verify specific properties of the model and the specification—assuming that the model actually models the specification. Beside pure modelling and verification, previous work has shown that executing the model is also essential: for understanding the model and evaluating simple test cases; for checking the model for "obvious" errors before embarking on a hard proof in a theorem prover; and finally for easily checking the validity of a given execution simply by running the model.

In case of the C++11 concurrency model, our concrete requirements for a tool chain are the following:

- HOL4 code for proof because of local expertise and to interface with previous work;
- Ocaml code to simulate the model and to use with the Cppmem execution checker;

– Isabelle/HOL code to interface with the Nitpick counterexample generator;
– finally readable LaTeX output for document preparation.

Naturally, adding another tool, such as Lem, to this already long list is not the only solution to this problem: one can formalise the model in Isabelle/HOL, for example, and handcraft small, task-specific tools to produce HOL4 input, to provide additional information to the Ocaml code generator in Isabelle, to fine-tune the LaTeX output, and so on.

The first iteration of our tool chain, which is used in [5], was created like this, but resulted in major caveats: eccentricities of each part in the tool chain forced sub-optimal design choices in the Isabelle definitions, for example the Isabelle to HOL translator could not handle bounded pairs in restricted quantifications; and many steps required additional, hand-written data, such as executable versions of definitions for the Isabelle code generator or auxiliary data to the HOLDoc type setting tool to produce readable LaTeX. In aggregate the tool-chain was brittle and unmaintainable; whenever a change to the model was made, it required pervasive changes to many different files.

In the following, we detail key features of Lem's specification language using the Lem code of our C++ memory model. We focus on features which are not necessarily in the familiar intersection of HOL and functional programming, but features which address our main motivation points and make Lem an adept tool for formalising large-scale semantics.

3.1 Backend-specific Definitions

As mentioned before, Lem's syntax is not restricted to the strict intersection of all the target languages; such a restriction, like no support for inductive relations as they are not supported by Ocaml, would make the tool often practically unusable as a formal specification system.

In our C++ model, the rules of the memory model are stated as predicates on relations between memory loads and stores, some of which use transitive closure. Unfortunately, there is no *elegant* transitive closure definition that lives in the intersection of Ocaml, Isabelle/HOL, and HOL4: the natural way of defining a transitive closure operator in Ocaml uses recursion. The recursive definition of a transitive closure, however, cannot be used in the theorem provers since there is no general termination proof without additional assumptions.

Lem supports two concepts to solve this problem without any hand-editing of generated code: backend-specific definitions and substitutions. In the former case, the user simply parametrises a standard `let` definition or inductive relation definition with a backend identifier. In the later case, the user provides a substitution rule that maps a local Lem identifier to a target-specific identifier in Lem's target library. We illustrate the use of both concept using the transitive closure definition of the C++ model.

In order to provide backend-specific definitions, the type of the identifier has to be defined first. Lem expects such a type specification when multiple definitions for a single identifier are made.

```
val tc : forall 'a. ('a * 'a) set -> ('a * 'a) set
```

In our C++ model, we choose to use a recursive definition not only for the OCaml code, but also for generated LaTeX code:

```
let rec tc r =
  let one_step = { (x,z) | forall ((x,y) IN r) ((y',z) IN r) | y = y' } in
  if one_step subset r then r else
  tc (one_step union r)
```

Note, that this definition does is neither backend-specific nor a substitution: Lem supports the use of a *default* definition which is used unless for there is a backend-specific definition for the currently processed target.

Since HOL4 has a transitive closure definition in its standard library, we define a substitution to use the library definition in the generated HOL4 code.

```
sub [hol] tc = tc
```

It is important to note the semantics of a `sub [target]` statement: when Lem processes such a statement, it opens its `target` library and *maps* the right-hand side of the equation into the library scope. In this case this simply means that Lem's HOL library has to contain a type definition for `tc`:
(excerpt from set_relation.lem, not in the C++ code)

```
type 'a reln = ('a * 'a) set
val tc : forall 'a. 'a reln -> 'a reln
```

Using substitution, the generated HOL4 code uses the transitive closure definition form its own library, which allows the re-use of already proven properties in the prover code.

Finally, to illustrate backend-specific definitions as well, we state the transitive closure definition for Isabelle/HOL using inductive relations; we intensionally do not use an Isabelle/HOL standard library function here.

```
indreln {isabelle}
  forall r x y. r (x, y) ==> tc' r (x, y) and
  forall r x y. (exist z. tc' r (x,z) && tc' r (z,y)) ==> tc' r (x,y)

let {isabelle} tc r =
 let r' = fun (x,y) -> ((x,y) IN r) in
 { (x,y) | forall ((x,y) IN r) | tc' r' (x,y) }
```

3.2 Minimal Perturbation in Translation

As previously mentioned, preserving the source layout is one of the main design goals of Lem: the layout, such as indentation and spacing, is usually intentional and meant to improve readability. Therefore, the Lem parser is able to parse any auxiliary, semantically irrelevant parts of the source code, such as comments, (redundant) white spaces, and line breaks, and stores them in the abstract syntax tree with the remaining code.

This ensures that Lem output looks very similar to the original source, no matter which backend is used. This is a crucial aspect of Lem: only this way, the generated code is as readable as the source, and one can work in a prover using the generated code only without needing the Lem source as a reference. The following, properly indented and commented, excerpt from the C++ Lem source is used as an example here.[1]

```
(* CoRR *)
( forall ((x,a) IN rf) ((y,b) IN rf).
  ((a,b) IN hb && same_location a b && is_at_atomic_location lk b) -->
    ((x = y) || (x,y) IN mo) ) &&
(* CoWR *)
( forall ((a,b) IN hb) (c IN actions).
  ((c,b) IN rf && is_write a && same_location a b && is_at_atomic_location lk b) -->
    ((c = a) || (a,c) IN mo) ) &&
(* CoRW *)
( forall ((a,b) IN hb) (c IN actions).
  ((c,a) IN rf && is_write b && same_location a b && is_at_atomic_location lk a) -->
    ((c,b) IN mo) )
```

The corresponding fragments in the target languages (in order: HOL4, Isabelle, OCaml, LaTeX) look very similar making the code as legible as the source.

```
(* CoRR *)
( ! ((x,a) :: rf) ((y,b) :: rf).
  ((a,b) IN hb /\ same_location a b /\ is_at_atomic_location lk b) ==>
    ((x = y) \/ (x,y) IN mo) ) /\
(* CoWR *)
( ! ((a,b) :: hb) (c :: actions).
  ((c,b) IN rf /\ is_write a /\ same_location a b /\ is_at_atomic_location lk b) ==>
    ((c = a) \/ (a,c) IN mo) ) /\
(* CoRW *)
( ! ((a,b) :: hb) (c :: actions).
  ((c,a) IN rf /\ is_write b /\ same_location a b /\ is_at_atomic_location lk a) ==>
    ((c,b) IN mo) ))‘;
```

Lem does not generate any x-symbol ASCII strings for operators in Isabelle, such as `\<and>` for `&`, to preserve readability of the code in its standard ASCII representation.

```
(* CoRR *)
(( ALL (x,a) : rf.  ALL (y,b) : rf.
  ( Set.member (a,b)  hb & same_location a b & is_at_atomic_location lk b) -->
    ((x = y) | Set.member  (x,y)  mo)) ) &
(* CoWR *)
(( ALL (a,b) : hb.  ALL c : actions.
  ( Set.member (c,b)  rf & is_write a & same_location a b & is_at_atomic_location lk b) -->
    ((c = a) | Set.member  (a,c)  mo)) )
(* CoRW *)
(( ALL (a,b) : hb.  ALL c : actions.
  ( Set.member (c,a)  rf & is_write b & same_location a b & is_at_atomic_location lk a) -->
    ( Set.member (c,b)  mo)) )"
```

In Ocaml, quantifiers are written without syntactic sugar and the logically equivalent disjunction is used instead of implication:

```
(* CoRR *)
( Pset.for_all (fun (x,a) -> Pset.for_all (fun (y,b) -> (not
  ( Pset.mem (a,b)  hb && (same_location a b && is_at_atomic_location lk b)) ||
    ((x = y) || Pset.mem  (x,y)  mo))) rf) rf ) &&
```

[1] We use a small font size for these examples to avoid having to add line breaks.

```
(* CoWR *)
(( Pset.for_all (fun (a,b) -> Pset.for_all (fun c -> (not
  ( Pset.mem (c,b)  rf && (is_write a && (same_location a b && is_at_atomic_location lk b))) ||
    ((c = a) || Pset.mem  (a,c)  mo))) actions) hb ) &&
(* CoRW *)
( Pset.for_all (fun (a,b) -> Pset.for_all (fun c -> (not
  ( Pset.mem (c,a)  rf && (is_write b && (same_location a b && is_at_atomic_location lk a))) ||
    ( Pset.mem (c,b)  mo))) actions) hb ))
```

Also in the LaTeX output, comments, line breaks and indentation match the source:

```
let coherent_memory_use actions lk rf mo hb =
  (* CoRR *)
  (∀(x, a)∈rf (y, b)∈rf.
    ((a, b) ∈ hb ∧ (same_location a b ∧ is_at_atomic_location lk b)) ->
      ((x = y) ∨ ((x, y) ∈ mo))) ∧
  (* CoWR *)
  ((∀(a, b)∈hb c∈actions.
    ((c, b) ∈ rf ∧ (is_write a ∧ (same_location a b ∧ is_at_atomic_location lk b))) ->
      ((c = a) ∨ ((a, c) ∈ mo))) ∧
  (* CoRW *)
  (∀(a, b)∈hb c∈actions.
    ((c, a) ∈ rf ∧ (is_write b ∧ (same_location a b ∧ is_at_atomic_location lk a))) ->
      ((c, b) ∈ mo)))
```

The output is faithful enough to the Lem source to allow proof on generated sources. Indeed, in work submitted for review [4] we use Lem generated HOL4 as base definitions for proofs. Our main theorem shows the redundancy of a particularly complicated rule which specifies the values of a memory read using existential quantification over a set of sets of memory writes; omitting this rule results in a significantly simpler model which is equivalent to the complex one, where equivalence here refers to both models producing the same set of executions for every C++11 program.

4 Implementation

Lem is written in OCaml, and it loosely follows the architecture of a traditional compiler. The central data structure is a typed abstract syntax tree (AST), which Lem builds and processes in 6 steps:

1. lex and parse the source files into untyped ASTs;
2. type check the ASTs, and convert them into typed ASTs;
3. transform the typed ASTs with target-specific macros;
4. rename variables and top level definitions, as required by the target;
5. add extra parentheses and remove infix operators, as required by the target;
6. print the resulting AST in the target's syntax.

Lexing and parsing Lem's input parser is built with `ocamllex` and `ocamlyacc`, which are OCaml's implementations of the Lex and Yacc lexer and parser generators. Lem's lexer and parser depart from standard practice by preserving all spaces, line breaks and comments of the input, so that they can be included

in the output. The lexer tracks these characters and returns them with their following token. The parser then incorporates them into the untyped AST. For example, when processing the following definition, the comment `(* Define x *)` and a line break are returned with the `let` token. Furthermore, a single space is returned with each of `x` and `=`, and a line break and two spaces are returned with the `1` token.

```
(* Define x *)
let x =
  1
```

The OCaml type declarations for the untyped AST are automatically extracted from the formal definition of Lem's syntax by Ott, to help us keep the implementation and formal specification in agreement. We also test Lem's handling of spaces by checking that parsed (and type checked) input can be converted to character-by-character identical output.

Type checking Lem's type checker uses a standard approach based on constraint unification [34, Section 3], where a constraint either requires two types to be equal, or requires a type to be an instance of a type class. No local constraints are used, and generalization only occurs for top-level definitions (e.g., nested `let`s are not polymorphic), so a single global store of type class instantiation information is sufficient. Constraints are required to have unique solutions, so that no type can be a member of a particular class in multiple ways. Instantiations can give rise to implicational constraints, e.g., type `'a list` is a member of the `Eq` class whenever its parameter `'a` is.

The implementation generally follows the formal type system (which is specified in Ott); however, it must also disambiguate some nodes of the untyped AST. The '`.`' character is used both for projection from modules and records, and so type information must be used to determine which is which. This differs from OCaml where a variable referring to a record must start with a lower case letter, and a module name must start with an upper case one, allowing for a purely syntactic disambiguation. In Lem, variables can start with either upper or lower case letters.

If the input is well typed, the type checker produces a AST that maintains all of the lexical information on whitespace and comments, and also annotates every sub-expression with its type.

Transformation The various output targets do not support all of the features of Lem. Thus, Lem must remove any unsupported expression or definition forms and replace them with equivalent supported ones. It does this for each output target separately. For example, OCaml does not support set or list comprehensions, and so those are translated into explicit iterations using fold operators (as long as the comprehension variables are given explicit ranges; the translation must fail otherwise). In HOL4, set comprehensions are supported, but list comprehensions are not, so only the latter is translated.

The transformation system is structured as a LISP-style macro expander. That is, it makes a top-down pass over each expression and at each subexpression

it checks if there are any macros that apply. If there are, it applies the first to get a new subexpression, and then repeats the check and apply until there are none. It then continues the traversal using the newly generated subexpression. This design allows each transformation to be written separately (and so easily used in different combinations for different back ends), and without repeatedly writing AST traversal code. Because it operates on the typed AST, the result of each macro is required to have the same type as its input.

Dictionary passing Neither HOL4 nor OCaml support type classes, and so Lem must be able to remove them. It uses a simple dictionary passing translation [18] where each function with a constrained type parameter gets an extra "dictionary" argument that contains implementations of the class' functions (e.g., the `=` function of the `Eq` class) for the parameterised type. Because constructor classes (e.g., monads) are not supported, the dictionary itself is typeable in Lem's type system. In cases where a class member function is called on a concrete type, no dictionary is needed. For example, in the function `let f x = (x = 2)`, the implementation of `=` for numbers can be referred to directly without giving `f` an extra argument.

Substitution The standard libraries of the various targets have differing data representations and interfaces, but to avoid the proliferation of back-end specific specifications, Lem only supports a single interface for each desired feature (e.g., finite maps, or bit vectors). Lem's standard library specifies how to translate from its own representation to each back end using substitution functions that are inlined during the transformation process. For example, the following excerpt from the list library specifies that each occurrence of `List.length` translates to `LENGTH` in HOL4, and to `List.length` in Isabelle/HOL and OCaml.

```
val length : forall 'a. 'a list -> num
sub [hol] length = LENGTH
sub [ocaml] length = List.length
sub [isabelle] length = List.length
```

Each of these substitutions is type checked using Lem types for the various target system libraries to ensure that the substitution generates type-correct code. This improves on Ott's similar *hom* functionality, which performs target specific substitutions, but does no type checking, possibly leading to output code with type errors whose source was not apparent in output since it arose from a bad substitution.

Renaming Although Lem, OCaml, HOL4, and Isabelle/HOL have similar scoping rules, and are all based on λ-calculus, they have significant differences in how names are treated. Furthermore the macro transformation system is not hygienic or referentially transparent [11,20], possibly leading to inadvertent name capture or collision. A target-specific renaming pass ensures that each variable will refer to the correct local variable or top-level definition when when the target system processes the generated output file.

For example, top-level names become special in HOL4 and Isabelle/HOL and cannot be used as parameters in function definitions. Thus, the following definition is invalid:

```
let x = 1
let add1 x = x + 1
```

and must be renamed:

```
let x = 1
let add1 x0 = x0 + 1
```

Similar trouble can occur for HOL4 if we use `LENGTH` as a parameter name.

```
let eq_length l LENGTH = (List.length l = LENGTH)
```

This must be renamed so that the `LENGTH` function that comes from the library substitution of `LENGTH` (see above) is not captured by the local variable.

```
let eq_length l LENGTH0 = (LENGTH l = LENGTH0)
```

Lastly, HOL4 and Isabelle/HOL do not support nested modules, and so they must be flattened while avoiding the creation of conflicting names.

Parenthesis insertion Just as the target systems have subtly different scoping rules, they also have different parsing rules. Lem contains a model of the precedences that each target system assigns to its infix operators, and it adds extra parentheses wherever the output would not be parsed correctly. It avoids adding unnecessary parentheses, so that the output will resemble the input as much as possible. This step happens after the transformation step so that the macros need not avoid introducing infix operations.

Printing All of the complex processing is performed on typed abstract syntax trees in the above stages, keeping the printing step as simple as possible, and somewhat uniform across the various target. Instead of using a pretty printing algorithm for layout, Lem uses the spacing from the input. Furthermore, a model of the lexical structure of the target is used to insert extra spaces wherever required. This frees the macro transformations from having to worry about whether to insert spacing or not.

5 Future work

We are actively developing Lem: our immediate goal is to polish the existing backends (and finish the in-progress Coq backend). Subsequently, we plan to add more backends, e.g., for Haskell or ACL2, and to re-engineer Ott to use Lem as a translation target, as mentioned above. This way, Off's current ad-hoc code generator for Isabelle/HOL, Coq, and HOL4 can be replaced. In order to increase the Lem code base and to use existing work more efficiently, we are also planing on looking at translators to import existing code to Lem: a HOL4-to-Lem translation—allowing us to automatically port, for example, Fox's

detailed ARM instruction semantics [13] to other provers—as well as 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 4.3 (2011), `http://www.cs.utexas.edu/~moore/acl2/`
2. Alkassar, E., Paul, W., Starostin, A., Tsyban, A.: Pervasive verification of an OS microkernel: Inline assembly, memory consumption, concurrent devices. In: VSTTE 2010. LNCS, vol. 6217, pp. 71–85. Springer, Edinburgh, UK (2010)
3. Aydemir, B.E., Bohannon, A., Fairbairn, M., Foster, J.N., Pierce, B.C., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S., Zdancewic, S.: Mechanized metatheory for the masses: The POPLmark Challenge. In: TPHOLs 2005. LNCS, vol. 3603 (2005)
4. Batty, M., Memarian, K., Owens, S., Sarkar, S., Sewell, P.: Clarifying and compiling C/C++ concurrency: from C++0x to power (2012), mauscript under submission
5. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. In: POPL 2011. pp. 55–66. ACM (2011)
6. 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)
7. 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)
8. Bishop, S., Fairbairn, M., Norrish, M., Sewell, P., Smith, M., Wansbrough, K.: Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations. In: POPL 2006 (2006)
9. Blanchette, J.C., Weber, T., Batty, M., Owens, S., Sarkar, S.: Nitpicking C++ concurrency. In: PPDP 2011. ACM (2011), to appear
10. Church, A.: A formulation of the simple theory of types. The Journal of Symbolic Logic 5(2) (Jun 1940)

11. Clinger, W., Rees, J.: Macros that work. In: POPL '91. pp. 155–162. ACM (1991)
12. The Coq proof assistant, v.8.3 (2011), http://coq.inria.fr/
13. 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)
14. F# (2011), http://msdn.microsoft.com/en-us/fsharp
15. Harrison, J.: HOL Light (2011), http://www.cl.cam.ac.uk/~jrh13/hol-light/
16. HOL 4, Kananaskis-7 (2011), http://hol.sourceforge.net/
17. Isabelle 2011 (2011), http://isabelle.in.tum.de/
18. Kiselyov, O.: Typeclass overloading and bounded polymorphism in ML (2007), http://okmij.org/ftp/ML/index.html#typeclass
19. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: SOSP '09. pp. 207–220. ACM (2009)
20. Kohlbecker, E., Friedman, D.P., Felleisen, M., Duba, B.: Hygienic macro expansion. In: LFP '86. pp. 151–161. ACM (1986)
21. Leinenbach, D., Petrova, E.: Pervasive compiler verification – From verified programs to verified systems. In: SSV 2008. ENTCS, vol. 217C, pp. 23–40. Elsevier (2008)
22. OCaml (2011), http://caml.inria.fr/ocaml/index.en.html
23. Owens, S.: A sound semantics for OCaml$_{light}$. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 1–15. Springer (2008)
24. Owens, S., Böhm, P., Zappa Nardelli, F., Sewell, P.: Lem: A lightweight tool for heavyweight semantics. In: ITP 2011. LNCS, vol. 6898, pp. 363–369. Springer (Aug 2011), ("Rough Diamond" section)
25. Peyton Jones, S. (ed.): Haskell 98 Language and Libraries: the Revised Report. Cambridge University Press (2003)
26. PVS 5.0 (2011), http://pvs.csl.sri.com/
27. 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)
28. Sarkar, S., Sewell, P., Alglave, J., Maranget, L., Williams, D.: Understanding POWER multiprocessors. In: PLDI 2011. ACM (2011), to appear
29. 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)
30. 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)
31. 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)
32. Strniša, R., Sewell, P., Parkinson, M.: The Java Module System: core design and semantic definition. In: OOPSLA 2007. pp. 499–514. ACM (2007)
33. Twelf 1.7.1 (2011), http://twelf.plparty.org/wiki/Main_Page
34. Vytiniotis, D., Peyton Jones, S., Schrijvers, T., Sulzmann, M.: OutsideIn(X) Modular type inference with local assumptions. Journal of Functional Programming FirstView, 1–80 (May 2011), DOI:10.1017/S0956796811000098