# Lem: reusable engineering of real-world semantics

Dominic P. Mulligan[1] Thomas Tuerk[1] Scott Owens[2] Kathryn E. Gray[1] Tom Ridge[3]
Peter Sewell[1]

[1] University of Cambridge [2] University of Kent [3] University of Leicester

## Abstract

Recent years have seen remarkable successes in *rigorous engineering*: using mathematically rigorous semantic models (not just idealised calculi) of real-world processors, programming languages, protocols, and security mechanisms, for testing, proof, analysis, and design. Building these models is challenging, requiring experimentation, dialogue with vendors or standards bodies, and validation; their scale adds engineering issues akin to those of programming to the task of writing clear and usable mathematics. But language and tool support for specification is lacking. Proof assistants can be used but bring their own difficulties, and a model produced in one, perhaps requiring many person-years effort, cannot be used by those familiar with another.

We introduce Lem, a language for engineering *reusable* large-scale semantic models. Lem definitions are translatable into OCaml for testing, Coq, HOL4, and Isabelle/HOL for proof, and LaTeX. This requires a delicate balance of expressiveness, careful library design, and implementation of transformations – akin to compilation, but subject to the constraint of producing usable and human-readable code for each target. Lem's effectiveness is demonstrated by its use in practice.

## 1. Introduction

Recent years have seen a rise in *rigorous engineering*: research projects making essential use of mathematically rigorous semantic models or specifications of key computational abstractions, such as processor architectures, programming languages, protocols, and security enforcement mechanisms. These models are used in many different ways: to elucidate the behaviour of an existing real-world abstraction, as oracles to test implementations against, as the underlying assumptions or goals of verification by mechanised interactive proof, as an explicit basis for program analysis, and as a medium for design. We recall a few representative examples to set the context for our work. Experimental semantics research, exploring existing real-world abstractions by a combination of empirical investigation and standards formalisation, has addressed the TCP/IP protocols [6], the sequential and concurrent behaviour of x86, Power, and ARM multiprocessors [9, 10, 29, 30], the (sequential) C standard [8], the concurrency model of C/C++11 [4], and JavaScript [7, 25]; these typically require models that can be used as oracles to decide whether some experimentally observed behaviour is permitted in the model, or to enumerate all the model-permitted behaviour. Verification work using mechanised interactive proof has produced verified compilers [18, 32, 36], verified operating system kernels [16], and verified secure fault isolation [22], each based on rigorous models of the underlying processor or language and of the abstraction the verified system aims to provide. Static and dynamic analysis must either build in implicit assumptions or be explicitly based on such a model, e.g. for binary analysis w.r.t. processor architectures [19].

This is a remarkable success story, contrasting with the state a decade ago when semantics was largely restricted to small idealised calculi: the above all involve models of (identified aspects of) real-world computer systems. But it raises its own problems. Constructing a substantial model is a major undertaking:

- one needs to understand the system being modelled in detail, often with experimental investigation of a de facto standard that has not been clearly specified even in prose;
- one has to deal with large-scale specifications, more like modest-scale programs (1–10k lines of specification) than the page or two of pencil-and-paper mathematics of a small calculus, with all the engineering and readability issues that entails; and
- models need extensive experimental validation, to establish confidence (in the absence of full formal verification down to the gate level) that they are correct models of reality; they need theoretical validation, e.g. by proofs of sanity properties showing that they are internally consistent; and they need social validation, discussion with the relevant vendors, standards committees, or community to ensure that they capture the right intent (especially for specifications that are looser than any particular implementation).

But language and tool support for such modelling and specification activities is lacking. As we discuss in more detail below, there has been no language that is specifically designed for the task, making model development more awkward than it should be, and, more importantly, *reuse* of models in different contexts is challenging and rarely achieved. For example, the above-cited works independently developed no less than six partial models of x86 instruction behaviour and two of JavaScript, and the literature contains yet more. For a small application-specific model this is not a problem, but where model development and validation may take person-years of effort such duplication of effort is not viable.

Our thesis is that, as the subject matures and rigorous engineering becomes more widespread, the community needs to amortise this effort, establishing a collection of models of the basic abstractions, those processor architectures, programming languages and protocols that are relatively stable interfaces that computer systems depend on. These should be comprehensive and well-tested, and they have to be made available in multiple forms to enable their use for many different purposes by different groups. This should lead to a virtuous circle: the prospect of reuse motivating more complete modelling and validation, and this enabling new research that would be impossible without substantial models.

In this paper we work towards this goal. Our main contribution is the design and implementation of a modelling and specification language, Lem, to support the engineering of reusable large-scale semantic models.

**Related work** To explain the distinctive features of Lem we first consider the alternatives. In some cases one can use a conventional typed functional programming language (such as Haskell, OCaml, or SML) to express a model as a pure functional reference implementa-

tion or test oracle. This gives the advantages of a mature and familiar programming language, but it does not give a basis for proofs about the model, and one often needs more logical expressiveness, especially for loose specifications. In particular, one needs simple syntax for sets and logic, inductive relation definitions, and a clearer understanding of when one is in the fragment of the language with a direct mathematical interpretation. For those, one typically turns to a proof assistant such as ACL2, Coq, HOL4, Isabelle/HOL, Matita or PVS. These provide very powerful proof tooling but are hard to master, and their definition languages have accreted functionality over time rather than being designed top-down as modelling/specification languages; inevitably introducing various idiosyncracies to the language.

More seriously, the community suffers from *proof-assistant lock-in*: the difficulty in becoming fluent in their use means that very few people can use more than one effectively, and the field is partitioned into schools around each. Indeed, even within some of our own projects we have had to use multiple provers due to differing local expertise. The differences between the tools mean that it is a major and error-prone task to port even the definition of a model from one to another, rarely attempted even where much effort has gone into model development. Sometimes this is for fundamental reasons: for example, definitions which make essential use of the dependent types of Coq may be hard or impossible to practically port to HOL4 or Isabelle/HOL. However, many of the examples cited above are logically undemanding: they have no need of dependent types, the differences between classical and constructive reasoning are not particularly relevant, and there is often little or no object-language variable binding. In such cases, where a model is basically expressible in the intersection of the definition languages of several proof assistants, it should in principle be possible to port definitions; the challenge is one of robustly translating between the source languages, definition styles, and libraries. This is made particularly hard by the sensitivity of proof assistants to whether definitions are *idiomatic*: given two logically equivalent definitions, one may be much more amenable than the other to machine-assisted proof or generation of executable code (for a test oracle) in a particular prover.

Previous work has established connections between different provers at the level of their internal logics [1, 11–14, 35]. These enable results proved in one system to be made available in another, but they do not provide usable *source definitions*. Between provers and programming languagues, all the provers mentioned above support some kind of code generation; the other direction is less developed, though Haskabelle [26] provides a mapping from a fragment of Haskell to Isabelle source.

**Contribution** Lem aims to combine the ease-of-use and uniform language design of programming languages with the logical expressiveness required for specification of the established proof assistants. Most importantly, it aims to support *portable* specifications, that can be used in multiple provers (it is not itself a proof tool). Spelling out our contribution in more detail:

*A language of executable mathematics* Lem is oriented towards (though not restricted to) executable definitions; definitions in the executable fragment of Lem can be translated into OCaml code to use as a test oracle for experimental validation, or for model exploration.

*Support for multiple proof-assistant targets* Lem definitions can be translated to proof-assistant definitions for Coq, HOL4, and Isabelle/HOL, to support interactive proof. The language design involves a delicate balance of expressiveness: enough for a range of large-scale modelling tasks, but still translatable into usable definitions in the various proof-assistant targets, as idiomatically as possible by automatic translation. These translations are related to, but interestingly different from, conventional programming-language implementation techniques; for example in the translations of equality and pattern matching. Perhaps surprisingly, in some cases it is best to translate into mathematically different code in different targets. The design and implementation choices that make this and the next point possible are described in §3.

*Human-readable output* It is also important to make the proof assistant generated definitions *human-readable*: Lem preserves the source structure and comments where it can (modulo the tension with generating idiomatic code), and it uses the same machinery to give the user control of layout for generation of production-quality LaTeX that can be used directly in papers and documentation, avoiding the error-prone and tedious manual typesetting of definitions for publication that can be necessary for some proof assistants. Lem can also generate simple HTML.

*Programming-language engineering* The language is designed using best-practice programming-language techniques, taking advantage of the opportunity to do a coherent design without the backwards-compatibility issues faced by proof assistants that have been extended over many years. The syntax and type system of Lem itself are specified using the Ott tool [31], which helped make the design regular (without odd corner cases); it should be easy to use by those familiar with typed functional languages such as OCaml, Haskell, or SML. Lem appears to the user much like a compiler: there is no need to learn a complex interface, and the implementation provides prompt feedback (e.g. for type errors) to the user, so that one can do type-based development and refactoring of specifications in the style of development in a typed programming language.

*Library design* A specification language needs a good standard library just as much as a programming language does. In §4 we describe the Lem library support and, more challenging, how it has to be related to the differing prover libraries.

*Substantial usage* Lem has been developed since 2010 and its effectiveness is demonstrated by a number of developments, with both academic and industrial impact. We begin in §2 by recalling those, to explain more clearly what it is (and is not) good for; we make several available in the Lem distribution. Lem is available from its Bitbucket source repository at `https://bitbucket.org/Peter_Sewell/lem/` under a BSD license.

*Note* Some of the motivation behind Lem, and its initial implementation, was initially presented in a short "Rough Diamond" paper [24]. Since then, Lem has been validated by substantial use (§2), and its design and implementation have become significantly more sophisticated (most of §3 and §4).

## 2. Lem in practice

To demonstrate the practical effectiveness of Lem as a tool for large-scale specification, we describe the main Lem developments produced to date. These underlie multiple academic publications (six in PLDI, POPL, and CAV) and have had industrial impact, in clarifying the IBM Power and ARM concurrency behaviour, on the C/C++11 ISO standard concurrency semantics, and on the compilation scheme of the latter to Power and ARM. For each development we give the number of non-comment lines of specification (los) and the Lem targets used. Snapshots of those marked ⋆ are at the URL above.

Sarkar et al. [29] describe an operational model (3008 los, ⋆) for the relaxed-memory behaviour of IBM Power and ARM multiprocessors. The executable Lem-generated OCaml code forms the kernel of the `ppcmem` tool for exhaustive and interactive exploration of the model on examples; that and the generated LaTeX supported experimental validation and extensive discussion with an IBM ar-

chitect during model development; and the generated Coq has been used for (unpublished) mechanised proofs.

Batty et al. [4] describe an axiomatic memory model (1517 los, ⋆) for concurrency in the C/C++11 standards. The Lem-generated OCaml and HTML form the kernel of the `cppmem` tool, again with a web interface for interactive and exhaustive exploration of the model on examples; that and the generated LaTeX supported discussion with the standards committee to develop the model and improve the standard. The generated HOL4 code has been used for mechanised proof of metatheory, and the generated Coq and Isabelle has also been used for (as yet unpublished) proofs.

Batty et al. [3, 28] describe extensions to the above models and correctness proofs for a compilation scheme from C/C++11 concurrency primitives to Power concurrency primitives; these are hand proofs but with lemmas expressed in Lem (931 los) (a useful middle ground between LaTeX and fully mechanised proofs).

Mador-Haim et al. [20] descibe an axiomatic model for Power and mappings between that and the operational model above (1155 los), again with hand proofs; the generated OCaml code was used to test equivalence of the two models on examples and the generated LaTeX to define the model in the paper. Unpublished work describes an operational model for ARM concurrency capturing some microarchitectural aspects (418 los), using the generated OCaml code to extend `ppcmem`.

The specification of the de facto standard of the TCP/IP network protocols and the Sockets API by Bishop et al. [6] was originally expressed in HOL4 and has now been ported to Lem (6681 los, ⋆).

The Ott definition of $\text{OCaml}_{\text{light}}$ by Owens [23], originally used to generate HOL4, has been ported to generate Lem (3133 los in Lem, from 4253 lines of Ott source ⋆).

Kumar et al. [17] describe a mechanically verified ML system above x86-64 machine code. The source language definition (parsing, AST, type checking, and small- and big-step operational semantics), compiler, and some additional machinery are in Lem (4897 los); the generated HOL4 code is used for proofs.

Work currently submitted for publication describes a model for a substantial part of the C programming language (8274 los).

Together, these demonstrate that Lem is expressive enough for a range of modelling tasks, spanning processor architectures, C and ML-like programming languages, and network protocols, and that it compiles to usable executable code for model exploration and usable definitions in multiple provers for proof. Anecdotally, the experience is that it is easy to use (analogous to a functional programming language) and that Lem models are malleable: in developing a model, one can largely focus on the domain being modelled rather than issues of expressing the model in Lem, and models can be easily changed as they are developed.

Lem is a general-purpose specification language but not, of course, an all-purpose one. It does not aim to support specifications with elaborate dependently typed hierarchies of mathematical structures. It has a straightforward syntax (again similar to that of a functional programming language), without support for rich user-defined syntax. The executable code Lem generates is designed to have a clear relationship to the source and has sufficient performance to support exploration of models on the intricate but small examples that (e.g.) arise as concurrency test cases; it is not aimed at producing performance-optimised code. Lem complements the Ott tool [31]: Ott supports arbitrary context-free user-defined syntax and inductive relations; this is a good fit for high-level programming language and calculus semantics but the lack of the more general types, functions and library support of Lem makes it awkward to use for modelling the lower-level systems and languages described above. Lem can serve as an intermediate language for other tools that produce definitions of types, functions, or inductive relations, and we have refactored Ott (which originally produced source code for Coq, HOL4, and Isabelle/HOL directly) to produce Lem definitions, leaving Lem to handle the prover-specific idiosyncrasies.

$$
\begin{array}{l}
typ ::= \ \_ \mid \alpha \mid typ_1 \rightarrow typ_2 \mid typ_1 \times \ldots. \times typ_n \\
\mid id\ typ_1 \ldots typ_n \mid backtick\_string\ typ_1 \ldots typ_n \mid (typ) \\
pat ::= \ \_ \mid (pat\ \textbf{as}\ x) \mid (pat : typ) \mid id\ pat_1 \ldots pat_n \\
\mid \langle\mid fpat_1; \ldots; fpat_n\ ;^{?} \mid\rangle \mid (pat_1, \ldots., pat_n) \mid [pat_1; \ldots; pat_n\ ;^{?}] \\
\mid (pat) \mid pat_1 :: pat_2 \mid x + num \mid lit \\
exp ::= \ id \mid backtick\_string \mid \textbf{fun}\ psexp \\
\mid \textbf{function}\ \mid^{?} pexp_1 \mid \ldots \mid pexp_n\ \textbf{end} \mid exp_1\ exp_2 \mid exp_1\ ix\ exp_2 \\
\mid \langle\mid fexps \mid\rangle \mid \langle\mid exp\ \textbf{with}\ fexps \mid\rangle \mid exp.id \\
\mid \textbf{match}\ exp\ \textbf{with}\ \mid^{?} pexp_1 \mid \ldots \mid pexp_n\ \textbf{end} \mid (exp : typ) \\
\mid \textbf{let}\ letbind\ \textbf{in}\ exp \mid (exp_1, \ldots., exp_n) \mid [exp_1; \ldots; exp_n\ ;^{?}] \\
\mid (exp) \mid \textbf{begin}\ exp\ \textbf{end} \mid \textbf{if}\ exp_1\ \textbf{then}\ exp_2\ \textbf{else}\ exp_3 \\
\mid exp_1 :: exp_2 \mid lit \mid \{exp_1 \mid exp_2\} \\
\mid \{exp_1 \mid \textbf{forall}\ qbind_1 \ldots qbind_n \mid exp_2\} \mid \{exp_1; \ldots; exp_n\ ;^{?}\} \\
\mid q\ qbind_1 \ldots qbind_n.exp \mid [exp_1 \mid \textbf{forall}\ qbind_1 \ldots qbind_n \mid exp_2] \\
\mid \textbf{do}\ id\ pat_1 \leftarrow exp_1; \ldots pat_n \leftarrow exp_n; \textbf{in}\ exp\ \textbf{end} \\
psexp ::= \ pat_1 \ldots pat_n \rightarrow exp
\end{array}
$$

**Figure 1.** Lem Syntax Excerpts

## 3. Design for portable specification

Aiming to support a range of specification tasks, Lem does not build in any domain-specific assumptions on the form of specification permitted: it is a general language of type, higher-order function, and inductive relation definitions. This contrasts with systems such as Ott [31] for inductive relations over inductive syntax, K [27] for rewriting, and PLT Redex [15] for reduction semantics.

The language is intended to be as expressive and straightforward as possible given this generality; it avoids novel or exotic features that would give it a steep learning curve, or render translations into the various targets infeasible. From functional programming languages we take pure higher-order functions, general recursion, recursive algebraic datatypes, records $\langle|\cdot|\rangle$, lists $[\cdot]$, pattern matching, parametric polymorphism, a simple type class mechanism for overloading, and a simple module system. To these we add logical constructs familar in provers: universal and existential quantification, sets $\{\cdot\}$ (including set comprehensions), relations, finite maps, inductive relation definitions, and lemma statements. Then there are facilities to let the user tune how Lem definitions are mapped into the various targets (by declaring target representations and controlling notation, renaming, inlining, and type classes), to generate witness types and executable functions from inductive relations, and for assertions. The concrete syntax for types and expressions broadly follows OCaml, as one can see in the excerpts in Fig. 1 (the main exceptions are prefix type applications, curried constructors, and records using $\langle|\cdot|\rangle$ to allow $\{\cdot\}$ for sets).

Although most of Lem's features should be unsurprising at first sight, their detailed design must carefully manage tradeoffs between Lem's expressiveness and usability as specification language, and the need to generate usable code for the various target languages and provers. In this section, we first describe the basic architecture of the Lem implementation, and then address how to design and implement these seemingly simple features – including polymorphism, equality, partiality, sets, and inductive relations – to satisfy those constraints. In the next section (Section 4), we will describe Lem's library definition mechanism, and explain how it connects the Lem standard library to the widely varying standard libraries of the targets.

### 3.1 System architecture

Lem is written in OCaml, and it follows the architecture of a traditional compiler invoked from the command line, with conventional lexing and parsing of source files into an untyped AST, followed by type inference (in the style of Milner's Algorithm W) into a typed AST. The OCaml type declarations for the untyped AST are automatically extracted from the formal definition of Lem's syntax by Ott [31], to help keep the Lem implementation and formal specification in agreement. Its parser and lexer are implemented using `ocamlyacc` and `ocamllex`.

To produce output for a particular target, the typed AST is then transformed, compiling away features that that target does not support (transforming away type classes via dictionary passing, compiling unsupported pattern matching, etc.). Special idiosyncrasies of the target may require additional clean-up (e.g., variable name clashes, extra required parentheses, different infix operator syntax); then the resulting AST is printed in the target's source syntax.

The transformation system is structured as a 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 step 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 reused 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, and this fact is checked by the implementation, greatly easing debugging.

Lem has about 24 000 lines of OCaml code. This compactness makes it easy to understand and adapt. The library is extensive in comparison to the code size. There are about 4 000 lines of Lem library files, 1 800 lines of OCaml, 1 200 lines of Isabelle/HOL, 400 lines of Coq and 300 lines of HOL4.

### 3.2 Polymorphism and dependency

Lem supports top-level parametric polymorphism. In a departure from the Hindley-Milner-style polymorphism found in functional programming languages, type generalisation is restricted to (module) top-level definitions. 'Let-polymorphism', the implicit generalisation of types to type-schemes in nested **`let`** bindings, is forbidden because it makes higher-order logic unsound. We have not found this to be limiting in practice, and Vytiniotis et al. [33, Section 4.3] provide empirical evidence that let-polymorphism is rarely used in practical Haskell programming.

More sophisticated type-language features, such as System F-style polymorphism, dependent types, and subtyping, are also not supported as these would be unduly difficult or impossible to support in many of our chosen targets. However, we do support *ad hoc* polymorphism with type classes.

### 3.3 Equality and type classes

There are substantial differences in the treatment of equality in our different targets. In the two implementations of higher-order logic, Isabelle/HOL and HOL4, there is a 'pervasive' equality constant $=$, at type $\alpha \to \alpha \to$ `bool`. OCaml features a similarly typed equality constant, but it is only usable for non-function types (raising an exception otherwise). Further, this OCaml polymorphic equality is structural, and does not take into account equivalence relations between data types. For abstract types such as sets (implemented in the Lem translation to OCaml as balanced binary trees) one needs to use an equality function specific to sets that compares sets based on their elements, rather than their low-level representation in memory, and that function must have access to the order relation used to build the trees. We could introduce specific equalities at each type, for example `setEq` at type $\forall\alpha.\ (\alpha \to \alpha \to$ `bool`$) \to$ `set` $\alpha \to$ `set` $\alpha \to$ `bool`. However, this would force the user to supply the equality function on elements of the set by hand (a task that should be automated), and break the uniformity of the treatment of equality within the language. Lem includes type classes to solve both of these problems.

The Lem `Eq` type class has the following form:

```
class (Eq α) =
  val (=) ['isEqual'] : α → α → bool
  val (<>) ['isInequal'] : α → α → bool
end
```

This type class introduces two methods, equality `=`, and inequality `<>` (together with alphanumeric alternative names). The type class may be instantiated at any type by providing implementations for the equality and inequality methods at that type.

For Coq, we extract Lem type classes to Coq type classes, leaving the responsibility for selection of the correct boolean equality function for a given type to Coq's type class instance search mechanism. For OCaml, Isabelle/HOL and HOL4, the Lem translation introduces explicit dictionary passing to handle the general case of type classes and their constraints. But there are three situations in which introducing dictionary passing would lead to non-idiomatic code and obstruct use of the extensive proof automation facilities of the provers.

First, we wish to map the overloaded equality constant of Lem to Isabelle/HOL and HOL4's native equality constant. We achieve this with a general inlining method:

```
let inline {hol;isabelle} (=)  =
  unsafe_structural_equality
let inline {hol;isabelle} (<>) =
  unsafe_structural_inequality
```

Here `unsafe_structural_equality` is the Lem equivalent of Isabelle/HOL and HOL4's equality constant. It should only be used by library implementors writing bindings to backends with a polymorphic boolean equality. This inlining effectively 'turns off' the equality type class for HOL4 and Isabelle/HOL. Since all methods are implemented without using the type-class mechanism, the class does not need to be generated for these backends.

Secondly, sometimes we want to use a method only for certain backends. Lem sets are represented in Coq and OCaml as ordered, balanced binary trees and therefore an order on their elements needs to be provided. This is achieved via a Lem type class `SetType`. However, the HOL4 and Isabelle/HOL sets do not require an order, so using the type class mechanism naïvely would lead to non-idiomatic HOL4 and Isabelle/HOL code by generating unnecessary dictionary arguments. By restricting class-methods to certain backends, this problem can be solved:

```
class ( SetType α )
  val {ocaml;coq} setElemCompare:  α → α → ordering
end
```

Lem's type-checker ensures that the method `setElemCompare` is only used in the Coq and OCaml backends, with the type class being safely eliminated for all other backends.

Thirdly, we wish to avoid introducing dictionary passing where all occurrences of a type class can be statically resolved, for example for the Lem type-class `Numeral`, used for overloading numeral syntax for multiple number types. This type class is declared as *inline*. All occurrences must be resolved, and all methods of an inlined type-class are replaced (inlined) with their instantiations.

Using type classes for equality in Lem also works around another issue in the Coq backend. In Coq the types `bool` and `Prop` (technically a *sort*) are distinct, with 'propositional equality' having type $\forall\alpha.\ \alpha \to \alpha \to$ `Prop`. It is generally impossible in

Coq, without additional axioms, to produce an equality constant of type $\forall\alpha.\ \alpha \to \alpha \to$ `bool`. Rather, 'boolean equalities' are given at specific types, such as `nateq` of type $\mathbb{N} \to \mathbb{N} \to$ `bool` and `booleq` of type `bool` $\to$ `bool` $\to$ `bool`, and so on.

Lem, following Isabelle/HOL and HOL4, identifies boolean-typed expressions and propositions, or formulae. This poses a problem with extraction to Coq, as innocuous Lem code—e.g. `if 4 = 5 then true else false`—becomes problematic if we try to extract an equality constant in Lem to Coq's propositional equality constant, as case analysis over `Prop` is not permitted. Boolean equality is therefore needed, and type classes are used to handle this uniformly.

Compared to Haskell's type class system, Lem's is less expressive—our goal is to support simple overloaded operators and to solve the above-mentioned issues dealing with the subtle differences between our target systems, not to enable generic and polytypic programming. In particular, Lem does not support constructor classes (type classes with type variables at a higher kind than $*$), default implementations of methods, or backchaining search for instances, nor does it support more recent, non-standardised extensions: multi-parameter type classes, functional dependencies, and so on.

Our implementation already has much of the underlying infrastructure needed to support instances at compound types, multi-parameter classes, default methods, and other features that would make type classes more convenient to use, and these should be easy to implement in the future. However, constructor classes and the like pose more significant technical challenges, because their dictionary-passing translations are not naturally typeable in the systems of ML-style polymorphism of our OCaml and HOL4 targets and Lem itself. One possibile design choice would be to support them, but only at statically known types (i.e., where dictionaries are not required).

### 3.4 Module system and 'do' notation

Lem has a simple module system designed to support the organisation of large-scale specifications into multiple files, and to allow the reuse of specification libraries (including Lem's standard library itself) across developments. It does not include the *programming-in-the-large* features of advanced PL module/component systems, such as enforced abstraction or parameterisation, because those are primarily useful in code bases that are orders of magnitude larger than even large specifications.

The module system is based on restricted subset of OCaml's module system: modules contain sequences of definitions, and modules can be defined at the top level or nested inside of other modules (but not inside of expressions). Definitions inside of a module are accessed from outside either through the `open` declaration, or by explicitly spelling out the module path (i.e., dot notation). There are no signatures or functors.

Lem supports a Haskell-like do notation for specifications that involve monads. Unlike Haskell, Lem's type class system is not powerful enough to infer which monad is being used (since a monad is a type constructor rather than a type). Instead, each `do` expression is annotated with a module name that defines the relevant bind and return operations.

### 3.5 Pattern matching

The usefulness of pattern matching is well-established in functional programming languages, and it is even more valuable in a specification language like Lem because it supports high level, abstract and clear code. But the support for pattern matching varies significantly between the different backends we target. For example, `as`-patterns are supported by OCaml and Coq, but not by Isabelle/HOL or HOL4. Record patterns are supported by OCaml, but not by HOL4, Isabelle/HOL or Coq. In contrast, idiomatic HOL4, Isabelle/HOL and Coq code uses pattern matching on natural numbers (using zero and the successor function as constructors), whereas this is not supported by OCaml. Besides differences in explicit pattern match expressions, there are also differences in how one can use patterns elsewhere. Anonymous functions in OCaml allow arbitrary patterns as arguments, but in HOL4, Isabelle/HOL and Coq only tuples of variables are allowed. The situation is similar for `let`-expressions and patterns occurring in restricted quantifications. Besides local, syntactic properties, semantic ones are important as well. Isabelle/HOL does support inexhaustive pattern matches, but no redundant rows. Coq requires pattern matches to be exhaustive and prohibits redundant rows. OCaml allows both redundant and inexhaustive pattern matches, whereas HOL4 allows inexhaustive pattern matches but redundant rows only at certain places.

We take the opportunity in the Lem language design to provide more general pattern matching, combining the facilities of each target, and compile those general patterns away where necessary. This compilation mostly follows a simple, standard approach, essentially switching on the outermost constructor symbols ([2]) in order to compile pattern matches to decision trees, implemented efficiently following ideas from [21]. But in contrast to the normal PL situation, where one wants to compile away pattern matching altogether, for Lem to produce idiomatic and human-readable code, we have to *preserve* as much of the original structure as possible; Lem needs sophisticated models of the capabilities of each backend in order to compile only the unsupported features and preserve as much of the original structure as possible.

For example, consider the following Lem record pattern match:

```
type t = ⟨| f1: nat; f2: bool |⟩

let test_fun x = match x with
  | Nothing ⇒ 0
  | Just ⟨| f2 = true |⟩ ⇒ 1
  | Just ⟨| f1 = 0 |⟩ ⇒ (1 : nat)
  | Just ⟨| f1 = x |⟩ ⇒ x + 2
end
```

OCaml supports all these pattern forms, so the resulting OCaml code looks very similar to the input; pattern compilation is not needed. In contrast, HOL4 does not support record patterns, and compiling them away – while preserving as much of the structure as possible – leads to the following HOL4 result:

```
val _ = Define ‘test_fun x = case x of
    NONE   ⇒ 0
  | SOME t ⇒ (case (t.f2, t.f1) of
        (T,_) ⇒ 1
      | (_,0) ⇒ (1 : num)
      | (_,_) ⇒ let x = t.f1 in x + 2)‘;
```

Our implementation also supports a mechanism similar to view patterns [34]. This powerful feature allows users to write even more abstract, higher-level code. For example, consider the type of sets. Functional programmers might be tempted to use the `choose` function to get the unique element out of a set known by the programmer to be a singleton. However, `choose` requires the axiom of choice in Coq, and its result is undefined for empty sets and underspecified for sets with more than one element. A solution is using a case-split for sets:

```
let set_case s c_empty c_sing c_else =
  if (null s) then c_empty else
  if (size s = 1) then c_sing (choose s) else
  c_else
```

This `set_case` can be implemented in all backends and is even executable. Lem's view-pattern feature allows setting it up together with `empty` and `singleton` for pattern matching.

```
declare pattern_match inexhaustive
  set 'a = [ empty; singleton ] set_case
```

This setup provides easily readable syntax, as below:

```
let set_test s : nat =
  match (s : set nat) with
    | empty ⇒ 0
    | singleton (x + 3) ⇒ 2
    | singleton _ ⇒ 1
    | _ ⇒ 3
  end
```

### 3.6 Partial pattern matches

Programming languages typically permit inexhaustive or partial pattern matches, with a dynamic exception or error if a match fails at runtime. This supports two use cases: (a) where the programmer knows that the match will never actually fail, e.g. because of some invariants that may not be expressible in the language's type system, and (b) where the intended control flow includes paths where an exception is raised and handled.

Proof assistants are more restrictive to retain soundness. For (a), in the HOL4 and Isabelle/HOL logics the Hilbert choice operator lets one construct an arbitrary unknown default value at any type, but in Coq all matches must be total (though the Coq type system can capture complex invariants). None of the three support (b) directly, as that would require a deeply embedded exception monad.

Lem permits partial matches, to support (a). For OCaml they are mapped directly onto similarly partial matches (any `Match_failure` exceptions can be handled by wrappers around the Lem-generated code, but not within that code). For HOL4 and Isabelle/HOL partial matches are mapped to syntactically partial matches that ultimately use that Hilbert choice operator, which is appropriate in cases where the match can never fail. For Coq, the story is more complex. Given a partial match at a concrete type, e.g.

```
match (m: maybe bool) with
  | Just j ⇒ j
end
```

that lacks a case for the `Nothing` constructor of the `maybe` type and whose result type is the concrete `bool`, we generate the following:

```
match (m: maybe bool) with
  | Just j ⇒ j
  | _ ⇒ bool_default
```

Here `bool_default` is a *default value*. Default values for base types are provided in an external harness file and Lem automatically generates default values for all user-defined types during generation of Coq code. Again this is suitable in cases where the user knows that the result of that branch is irrelevant.

Partial matches with a non-concrete result type are more problematic, e.g.

```
match (m: maybe β) with
  | Just j ⇒ j
end
```

Here the best we can do is translate into a complete Coq match

```
match (m: maybe β) with
  | Just j ⇒ j
  | _      ⇒ DAEMON (* From <position>. *)
end
```

that introduces a placeholder marking a point in the source specification that needs to be addressed, and a warning or error is issued. Here, `DAEMON` is a constant of type $\forall\alpha.\alpha$—the type of logical falsity. The use of `DAEMON` lets one build the remaining development but is dangerous from a logical perspective — its presence as an axiom makes Coq's logic inconsistent, so one would aim to remove all such usages. In practice we have not found many cases where this arises.

### 3.7 General recursion vs total functions

Functional languages typically allow general recursion, whereas proof assistants generally require some kind of termination proof for all recursive functions. For example, the function `let rec f x = not (f x)` will diverge in OCaml when called, but would introduce an unsoundness to a proof assistant. Defining a recursive function in Coq, HOL4 or Isabelle/HOL therefore requires the user, often assisted by the proof assistant itself, to supply evidence that the function terminates on all inputs. Lem can utilise the backend's automatic termination prover, where one exists, by declarations like

```
declare termination_argument my_rec_function = automatic
```

If that does not suffice, for the HOL4 and Isabelle/HOL backends Lem can defer termination proofs, letting the user provide them manually later (see §3.10).

In Coq the situation is more complex. Coq can spot that recursive functions exhibiting simple recursion schemes are terminating. If this machinery fails, the user is required to rewrite the function in terms of an accessibility predicate with an explicit well-founded order supplied. This requires more input from the user than the equivalent termination proofs of Isabelle/HOL and HOL4. Lem functions therefore must be written in a style Coq's termination prover can handle, and if not possible, the user must perform the transformation in Coq. In practice, this is seldom a problem and is greatly eased by Lem's backend-specific definition mechanisms.

### 3.8 Per-target representation differences

One might imagine that a portable specification should necessarily map onto mathematically equivalent definitions in the different targets, but this turns out not to be the case: there is a tension between it and the need to generate idiomatic code.

For example, in OCaml the standard type for numbers is `int`, 31- or 63-bit signed integers, while our proof assistant targets use unbounded natural numbers as their standard type. In each target common functions like the list `length` function use that target's local standard number type, so either Lem has to add wrappers to such functions or map the same Lem type to different mathematical constructs in different targets.

We give the user the choice, providing a Lem type `nat` which is translated to the standard number type of the targets (for use where the user knows the differences are irrelevant) and a Lem type `natural` which always maps to unbounded naturals. Similarly we provide `int` and `integer`. We provide conversion functions between the different number types; polymorphic numeral constants and polymorphic arithmetic functions make switching between number types easy.

The choice of set representations is more involved. Since Lem is geared towards executability, one might want to model only finite sets, and in practice users are often only concerned with finite structures, e.g. as arising in finite executions of the models described in §2. On the other hand, potentially infinite sets are very common and very useful for specifications and more convenient to work with in some provers. Similar questions occur with quantification. Only bounded quantification is easily executable, but unbounded quantification is often useful for specifications. Lem permits unbounded quantification and infinite sets, but only for the HOL4 and Isabelle/HOL targets; for OCaml and Coq it only support finite sets and bounded quantification. In future we will provide alternatives as for `nat` and `natural` above.

Having decided on finite or infinite sets, there remains a non-trivial choice of the best target representation. OCaml has a set

implementation in the standard libraries, but with a functorised interface to supply an order on elements; we use a library with a polymorphic interface instead. For Isabelle/HOL and HOL4 the idiomatic potentially infinite sets are used. Isabelle/HOL also has several interesting alternative ones with improved executability, but they are supplied with an order via Isabelle/HOL's type-class mechanism which would require the user to supply potentially nontrivial proof. For Coq, there is no single idiomatic set library: FSets provides a module-oriented implementation, the Collections wrapper around this library uses type classes, and one can also represent sets as functions into `bool` or `Prop` ('ensembles'). They all also require an order, or at least a decidable equality relation, which would have to be provided via Coq's type-class mechanism. We add a simple finite-set library, leaving the mapping of Lem sets to existing Coq libraries as future work.

Associative maps have similar issues to sets. However, for maps the idiomatic map-type in HOL4 is finite as well. Lem translates maps to Lem-specific finite map implementations in OCaml and Coq. For HOL4, we map into idomatic finite maps and for Isabelle/HOL to idiomatic infinite ones.

The relation type is closely related to sets and therefore has similar design issues. Relations can be seen as sets of pairs or as binary predicates. Both Isabelle/HOL and HOL4 provide dedicated libraries for both representations. Lem currently maps only to set representations. We leave the mapping to a predicate representation, in order to generate more idiomatic code, as future work.

### 3.9 Naming, notation and namespace issues

The set of pre-defined identifiers and reserved keywords is different in each backend. For example, `op` is a reserved word in Isabelle/HOL but not in Coq. Without reserving every reserved word and every pre-defined identifier in each of our backends in Lem, we must implement a renaming mechanism to avoid name clashes post extraction.

Lem maintains records of all of the reserved words of each backends. Moreover, there is a simple model of the namespaces of the backends. If a reserved word for the backend in question is encountered, Lem will automatically rename the constant and issue a warning. For example,

```
let op f g = ...
```

is automatically renamed to `op0`, or some other globally fresh name, when extracting to Isabelle/HOL. All occurences of this constant are also suitably renamed. A warning is issued on the command line to notify users of the renaming. The user also has full control over the renaming. For instance, placing

```
rename {isabelle} op = isaop
```

in a Lem source file will rename `op` to `isaop` during extraction to Isabelle/HOL. This mechanism allows us to avoid auto-generated names. Lem can thereby generate stable, predictable output even in the presence of name clashes. By renaming constants to follow the conventions of a particular backend the renaming feature also facilitates idiomatic backend code.

Note that the Lem renaming mechanism is smart enough to respect different lexical scopes. For example,

```
let op op = ...
```

features a function `op` taking a parameter called `op`. For some backends, this is problematic, and so the two should be renamed apart. In our running example, the function `op` will be renamed to `isaop` and the argument `op` to `op0` to avoid clashes with the Isabelle/HOL keyword. Lem correctly renames the name of the parameter apart from all constants present in the context.

### 3.10 Assertions, lemmata, and auxiliary outputs

Lem is intended to translate into target output that can be directly used, without manual editing by the user: Lem definitions generate target definitions which can be fully automatically checked. However, Lem can also generate aids for the user in additional *auxiliary* files, intended to be copied and manually edited by the user as necesary. For example, when defining a complicated recursive function in Lem, HOL4 and Isabelle/HOL can leave the termination proof to the user. Lem generates in the auxiliary file a template for its termination proof which the user can flesh out.

Lem also supports *assertions*, *lemmas* and *theorems*. Assertions are executable, for automated testing of simple properties. For OCaml they generate code in an auxiliary file that runs automated unit tests. For the theorem prover backends, they generate proof obligations, which are attempted to be discharged automatically. Lemmas and theorems are non-executable; they add proof obligations to auxiliary files. Simple, low-level lemmas can also be used for testing: often the resulting proof obligations can be automatically discharged. Exporting complicated lemmas and theorems might be beneficial as well. For example, Isabelle/HOL provides highly automated, powerful tools, and by exporting a lemma to Isabelle/HOL, this powerful machinery is easily accessible even by users not familiar with Isabelle. For example, the following Lem lemma

```
lemma unzip_zip:
  ∀l1 l2. unzip (zip l1 l2) = (l1, l2)
```

is translated to the Isabelle/HOL code:

```
lemma unzip_zip:
  "∀l1 l2. list_unzip (zip l1 l2) = (l1, l2)"
(* try *) by auto
```

The automated proof attempt by the `auto` method fails. If the user then uncomments `try`, various automated methods are run to either prove the lemma or find a counterexample. These methods include running external SMT and first order provers, internal natural deduction tools as well as a sophisticated counter example generator. In this example, Isabelle/HOL quickly finds a counterexample:

```
Nitpick found a counterexample for card 'a = 2 and card 'b = 2:
  Skolem constants: l1 = [a1], l2 = []
```

In general, tools like the counterexample generator Nitpick work well with Lem-generated Isabelle/HOL code, because Lem is tailored toward executability and this executability is (as far as possible) preserved by the translation to Isabelle/HOL. Therefore, non-trivial counterexamples can often be found automatically.

### 3.11 Inductive relations

Specifications often involve inductively defined relations, such as type systems or evaluation relations defined as the smallest relations satisfying a collection of rules. Lem provides an *inductive relations* mechanism for this purpose.

The following Lem definition captures the reduction relation of the call-by-value $\lambda$-calculus:

```
indreln [reduce: term → term → bool]
  ax_app: ∀ x t1 v2.
    (is_val_of_term v2) ⟹
      reduce (T_app (T_lam x t1) v2) (subst v2 x t1)
and
  ctx_app_fun: ∀ t1 t t1'.
    (reduce t1 t1') ⟹
      reduce (T_app t1 t) (T_app t1' t)
and
  ctx_app_arg: ∀ v t1 t1'.
    (is_val_of_term v) ∧ (reduce t1 t1') ⟹
      reduce (T_app v t1) (T_app v t1')
```

Here, `reduce` is introduced as a relation—Lem relations are essentially functions into `bool`—between AST `term`s. In this case, the `reduce` relation is defined via three clauses. Within a clause, the full power of the Lem language is available, rather than a purely relational subset, as in Prolog. For instance, `is_val_of_term` and `subst` are functions, rather than relations. One may also define mutually recursive inductive relations.

The Isabelle/HOL, HOL4 and Coq backends support inductive relations, and we map Lem inductive relations into the native inductive relations of these backends. (In the other direction, the above was actually generated by Ott from a similar definition expressed in a surface syntax for the calculus.)

However, inductive relations are not naturally expressible in our OCaml backend. We therefore implement a compilation process, compiling a Lem inductive relation into a function that searches for derivations. The compilation process is given a *mode* by the user, a description of which components of the relation are to be treated as 'inputs' and which are to be treated as 'outputs'. This compilation scheme is similar to one implemented within the Isabelle/HOL proof assistant, as implemented by Berghofer et al. [5] (there verified within Isabelle/HOL).

However, we go further than the Isabelle/HOL compilation scheme in automatically generating *witness types*, which encode a derivation tree for a given inductive relation. The functions generated by the compilation scheme can return witnesses, and additional functions check whether an element of the witness type belongs to the relation. These witnesses may also be produced externally, e.g. by a typechecker implementation that one wants to test, using the Lem-generated checker functions, against its definition. The generated types and functions are themselves defined in Lem and thus can be used by later Lem definitions and translated to any of the Lem targets.

For example, the following syntax instructs Lem to generate a reduction function for `reduce`, naming it `onestep`:

```
[reduce:  term  → term → bool
 onestep: input → output ]
```

The mode annotation on `onestep` instructs the compilation machinery to consider the first component of `reduce` as an input, and the second an output. Additional annotations can indicate that the function returns multiple results, that there must be a unique return value, or to generate a witness for the relation. Generated functions that are partial are treated in the same way as partial pattern matches. For non-deterministic rules, the generation searches exhaustively and, according to the annotation, either returns a list of elements in the relation or a single one (or is undefined).

To generate witness types and witness-checking functions, one can write:

```
[reduce : term → term → bool
 witness type r_witness; check check_r;]
```

This generates a witness type for the reduce relation:

```
type r_witness =
| Ctx_app_arg_witness of term × term × term × r_witness
| Ctx_app_fun_witness of term × term × term × r_witness
| Ax_app_witness of string × term × term
```

Instrumenting an interpreter or type-checker to produce such witnesses should be straightforward.

### 3.12 Prop and bool in the Coq backend

As mentioned in our discussion of equality and type classes in Lem, Coq maintains a distinction between a *sort* of propositions, `Prop`, and a *type* of boolean-valued expressions, *bool*. Lem, similarly to Isabelle/HOL and HOL4, collapses these two notions into a single type, `bool`. This mismatch between the languages causes difficulties, most notably in how we handle equality, as we have seen, but also in how we handle inductive relations and lemmata.

Take, as an example, the following Lem inductive relation:

```
indreln [even: nat → bool]
    even_zero: true ⟹ even 0
and even_plus: ∀n. even n ⟹ even (n + 2)
```

This is translated to an ordinary Coq inductive type residing in `Prop` (as all inductive types in Coq must reside in a sort):

```
Inductive even: nat → Prop :=
  | even_zero: true → even 0
  | even_plus: ∀n, even n → even (n + 2).
```

Here premises of the introduction rules of the inductive relation (for example, `true` in `even_zero`) are of boolean type, whereas they need to inhabit `Prop`. We use a Coq coercion from the Coq `bool` type into `Prop` to circumvent this problem: a function of type `bool` → `Prop` declared as a coercion automatically lifts a boolean expression into `Prop`. Type annotations are needed to guide this process, therefore extraction applies these to terms that should be lifted, sometimes redundantly. A similar problem occurs with lemma statements, which reside in `bool` in Lem but must reside in `Prop` in Coq. Again, we use the same solution.

However, problems persist elsewhere. One cannot perform a case analysis on `Prop` within Coq. However, Lem allows one to perform case analyses with `if`- and `case`-expressions on expressions whose images under extraction reside in `Prop`. For example, Lem allows users to perform a case analysis on inductive relations:

```
let odd n = if even n then false else true
```

In the extracted Coq code, the term `even n` has type `Prop` when it is expected to have type `bool` due to its position in the `if`-expression. There are three possibilities here:

1. Make this an error in Lem. However for backends like Isabelle/HOL and HOL4 the definition above is completely innocuous and rejecting it would be too restrictive.
2. Make use of Coq's generalised `if-then-else` notation (supporting any inductive type with exactly two constructors), include the `sumbool` type usually used to capture decidability, and attempt to automatically show decidability for inductive types such as `even` above. The above would then be translated to:

   ```
   let odd n = if even_dec n then false else true
   ```

   where `even_dec` is a decidability theorem of type $\forall n.\ \{\text{even } n\} + \{\neg \text{ even } n\}$. However, this approach becomes much more involved if we use inductive relations in more complex ways within Lem, for example, in a list of booleans: `[4 < 5, even 4, false]`, or in some other complex expression, and it is not clear how well this approach will scale.
3. Admit classical axioms—known to be consistent with Coq's logic—and collapse `Prop` into `bool`.

We ultimately adopt the last alternative, with a function `bool_of_Prop` of type `Prop` → `bool`, wrapping this function around any propositional term that is being used in a way where a boolean-typed term is expected. The admission of classical axioms to collapse `Prop` into `bool` is a matter of taste. Some large Coq developments happily assume classical axioms, others stay firmly within the existing constructive logic provided by Coq. We feel, however, that not restricting the Lem source language to accommodate every nuance exhibited by the backends is worth the admission of these axioms.

### 3.13 Whitespace preservation and refactoring support

Unlike a compiler—but similar to a refactoring tool—Lem's front end preserves all of the comments, whitespace, and line breaks in the source files. Lem attempts to format its output using the formatting of the input, rather than a pretty printing algorithm, in order to give the user fine-grained control of the shape of the output. Of course, this is not always possible when the input had to undergo significant transformation, such as the pattern match compilation or dictionary passing translation discussed earlier. In these cases, we use a standard pretty-printing algorithm for the affected expressions, but can at least keep all of the comments from the input. Crucially, the LaTeX backend does not perform such transformations, and so the user has control over the typesetting of their specifications, including linebreaks and indentation.

Since Lem preserves comments, whitespace and linebreaks, it can reproduce its input exactly. This means that Lem can easily be used as a powerful refactoring tool for it's own input. A special refactoring backend is able to rename functions and types, remove or add function arguments, move definitions to different modules and much more.

## 4. Library mechanisms and design

**Library design** The Lem distribution supplies a default set of types and functions in its library, focussed on specification. Collections such as lists, sets and maps, basic data types such as disjoint sums, optional types, booleans and tuples, useful combinators on functions, and a library for working with relations are all included.

Specific function names and types exposed to the Lem user by the library are adopted from the Haskell standard library where possible. This is a well-designed library with a focus on purity. We also wish to provide flexibility in the specific choice of *backend* libraries function names and types in the Lem library are mapped to. Often, picking one library over another involves a trade-off between competing factors, with no clear winner. We therefore made it possible for users to change and extend the library as they see fit, including replacing it wholesale. Lem library files are standard source files, their only distinguishing feature being their inclusion in the Lem distribution. No 'prelude' or 'pervasive' environment is automatically loaded during Lem compilation.

Further, proof assistants and programming languages differ in how partiality and other computational effects are handled. Proof assistants also implement different logics. Whilst Coq is a constructive type theory, Isabelle/HOL and HOL4 are both based on classical higher-order logic extended with Hilbert's choice—a principle Coq lacks without additional axioms.

Lem aims to accommodate both programming languages and proof assistants as backends. Further, we aim to support users who wish to target a subset of these backends, or all of them, favouring neither one nor the other. As a result, the Lem library must be suitably flexible in its design. We bifurcate the library into two sets of modules: the 'main' and 'extra' modules. The main hierarchy of files contain total, terminating functions that we believe are well-specified enough to be portable across all backends. All other functions are are placed in the extra modules. For example, the library file `function.lem` includes various useful combinators such as `flip` and `const`. The `function_extra.lem` file, on the other hand, contains the constant `THE` with type $\forall\alpha.\ (\alpha \to \texttt{bool}) \to \texttt{maybe}\ \alpha$, inexpressible in Coq.

Our design philosophy in the library is *permit partiality and under-specification, but isolate them*. The 'main' and 'extra' partition of library modules, combined with the fact that the user must always explicitly import the library files they are using, means there is always a conscious decision made on the part of the user to import functionality that assumes choice or exhibits partiality into their development.

**Technical mechanisms** Proof assistants provide a large body of facts about data types such as lists and numbers. Often, these facts are bundled together into simplification procedures for use in proof automation. In line with our goal of producing idiomatic backend code, we would like to map data types and functions in our Lem source to their corresponding implementations in the proof assistant, rather than extract our own copies. Towards this end, Lem features an array of tools for binding Lem functions and types to existing functions and types in the backends. For example

```
declare ocaml target_rep type set = 'Pset.set'
```

declares that Lem sets should be represented in OCaml by the existing OCaml-type `PSet.set`. Similarly, constants and functions can also be mapped to existing target representations:

```
val snoc : ∀α. α → α → list α
let snoc e l = l ⊕ [e]
declare hol target_rep function snoc = 'SNOC'
let inline {isabelle;coq} snoc e l = l ⊕ [e]
```

Here, we introduce a Lem constant `snoc` and provide a Lem definition for it. For the HOL4 backend, `snoc` is mapped to the `SNOC` function from the HOL4 list-library. Coq and Isabelle/HOL do not provide their own native constants and this operation is expressed idiomatically using list concatenation. Lem's inlining mechanism replaces all occurrences of the `snoc` constant with the append of a singleton list in the extracted Isabelle/HOL and Coq code. Finally, for all other backends that Lem supports (i. e. OCaml in the example), a default implementation of the function is extracted. Note no inlining occurs here—the call to `snoc` is preserved, with the resulting OCaml code looking similar to the Lem source code.

The Lem target-representation and inlining mechanisms are powerful enough to 'smooth over' inconsistencies between the backends. For instance, folds over lists display a surprising variety in the order in which arguments are expected. Our mechanisms allow us to provide a consistent interface within Lem for functions such as these whilst mapping to idiomatic backend code:

```
declare hol   target_rep function foldr = 'FOLDR'
declare ocaml target_rep function foldr f b l =
  'List.fold_right' f l b
```

Target representations can also declare a constant as infix for certain backends. For example, the set-membership constant is mapped as follows:

```
declare ocaml target_rep function member = 'Pset.mem'
declare hol   target_rep function member = infix 'IN'
declare html  target_rep function member = infix '&isin;'
```

This function is prefix for OCaml but infix for HOL4. We could also provide additional associativity and binding strength information about infix constants in order to avoid generating superfluous parenthesis during extraction. Note that we can also provide HTML and LaTeX target representations and that target representations are not restricted to valid Lem identifiers.

**Unit testing through lemmata** The Lem library attempts to clarify the semantics of its functionality by providing a definition, even if there are target-specific representations for all targets. Moreover, assertions and lemmata statements can be used to describe the supposed behaviour of library files. As described in Section 3.10, assertions are executable tests used for unit testing. They generate executable testing code for OCaml. For the theorem prover backends they generate proof obligations which are (mostly) closed automatically by the prover's computation mechanisms. Lemmata are non-executable tests. They are ignored by the OCaml backend,

while the theorem prover backends generate proof obligations that need to be discharged manually by the user. For the `snoc` example, the library contains the following assertions and lemmata:

```
assert snoc_1 : snoc (2:nat) [] = [2]
assert snoc_2 : snoc (2:nat) [3;4] = [3;4;2]
lemma snoc_length :
   ∀e l. length (snoc e l) = succ (length l)
```

If both a definition and a target-specific representation are present, Lem automatically generates a lemma that the target representation satisfies the definition. For example, the following lemma is automatically generated for the HOL4 backend:

```
lemma snoc_def_lemma: ∀l e. (l ⊕ [e]) = (SNOC e l)
```

Whilst we do not immediately aim to completely describe the semantics of every function in the Lem library with assertions and lemmata, we believe this peppering of executable checks and proof obligations provides some assurance that each of the bindings in the respective backends has the intended semantics.

## 5. Conclusion

Lem provides a new alternative for building large-scale semantic models and specifications, combining the uniform language design and ease of use of a good programming language with the definitional expressiveness provided by a theorem prover, and supporting *portable* definitions. It is more general-purpose than existing specification languages like K, Ott, or PLT Redex. Using Lem inevitably imposes some restrictions compared with working natively in a single prover but offers some advantages even in that case, and for modelling/specification exercises where the model creation and validation effort is large, the prospect of portability is compelling. It is demonstrably flexible enough to naturally specify a wide range of large-scale models while also allowing users to use their preferred tools for proof.

Lem's syntax and type system are formally defined, but its logical semantics is defined by the translations into the targets. We attempt no formal guarantee that the result of translating into one target has the same mathematical meaning as that of translating into another, and indeed sometimes they intentionally do not (§3.8). Even when intuitively they do, stating and proving that fact would require creating, as a starting point, formal models of the semantics of the various targets, including Coq's underlying type system, Isabelle's datatype package, HOL's inductive relations package, etc. That would be a worthy and challenging goal, but ours here is pragmatic: to support the working specifier.

**Acknowledgements** We thank Thomas Williams for his work on the Lem inductive relations package, Ohad Kammar for his work on model porting, and all the users of Lem. We acknowledge funding from EPSRC grants EP/H005633 (Leadership Fellowship, Sewell) and EP/K008528 (REMS Programme Grant).

## References

[1] A. Asperti, C. Sacerdoti Coen, E. Tassi, and S. Zacchiroli. User interaction with the Matita proof assistant. *Journal of Automated Reasoning*, 2006.

[2] L. Augustsson. Compiling pattern matching. In *Functional Programming Languages and Computer Architecture*, LNCS 201. 1985.

[3] M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell. Clarifying and compiling C/C++ concurrency: from C++11 to POWER. In *Proc. POPL*, 2012.

[4] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *Proc. POPL*, 2011.

[5] S. Berghofer, L. Bulwahn, and F. Haftmann. Turning inductive into equational specifications. In *Proc. TPHOLs*, 2009.

[6] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and Sockets. In *Proc. SIGCOMM*, 2005.

[7] M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith. A trusted mechanised JavaScript specification. In *Proc. POPL*, 2014.

[8] C. Ellison and G. Rosu. An executable formal semantics of C with applications. In *Proc. POPL*, 2012.

[9] A. C. J. Fox and M. O. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *Proc. ITP*, 2010.

[10] S. Goel, W. A. Hunt Jr., and M. Kaufmann. Abstract Stobjs and their application to ISA modeling. In *Proc. ACL2 Workshop*, 2013.

[11] M. J. C. Gordon, J. Reynolds, W. A. Hunt Jr., and M. Kaufmann. An integration of HOL and ACL2. In *Proc. FMCAD*, 2006.

[12] J. Hurd. The OpenTheory standard theory library. In *NASA Formal Methods*, LNCS 6617, 2011.

[13] C. Kaliszyk and A. Krauss. Scalable LCF-Style proof translation. In *Proc. ITP*, LNCS 7998, 2013.

[14] C. Keller and B. Werner. Importing HOL Light into Coq. In *Proc. ITP*, LNCS 6172, 2010.

[15] C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J. A. McCarthy, J. Rafkind, S. Tobin-Hochstadt, and R. B. Findler. Run your research: on the effectiveness of lightweight mechanization. In *Proc. POPL*, 2012.

[16] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an OS kernel. In *Proc. SOSP*, 2009.

[17] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: A Verified Implementation of ML. In *Proc. POPL*, 2014. To appear.

[18] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[19] Junghee Lim and Thomas Reps. TSL: A system for generating abstract interpreters and its application to machine-code analysis. *ACM TOPLAS*, 35(1), April 2013.

[20] S. Mador-Haim, L. Maranget, S. Sarkar, K. Memarian, J. Alglave, S. Owens, R. Alur, M. M. K. Martin, P. Sewell, and D. Williams. An axiomatic memory model for POWER multiprocessors. In *CAV*, 2012.

[21] Luc Maranget. Compiling pattern matching to good decision trees. In *Proc. Workshop on ML*, 2008.

[22] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan. RockSalt: better, faster, stronger SFI for the x86. In *Proc. PLDI*, 2012.

[23] S. Owens. A sound semantics for OCaml light. In *Proc. ESOP, LNCS 4960*, 2008.

[24] S. Owens, P. Böhm, F. Zappa Nardelli, and P. Sewell. Lem: A lightweight tool for heavyweight semantics. In *Proc. ITP, LNCS 6898*, pages 363–369, 2011. "Rough Diamond" section.

[25] J. G. Politz, M. J. Carroll, B. S. Lerner, J. Pombrio, and S. Krishnamurthi. A tested semantics for getters, setters, and eval in JavaScript. In *Proc. DSL*, 2012.

[26] Tobias Rittweiler and Florian Haftmann. Haskabelle — converting Haskell source files to Isabelle/HOL theories. http://isabelle.in.tum.de/haskabelle.html.

[27] G. Roşu and T. F. Şerbănuţă. An overview of the K semantic framework. *J. Logic and Algebraic Programming*, 79(6):397–434, 2010.

[28] S. Sarkar, K. Memarian, S. Owens, M. Batty, P. Sewell, L. Maranget, J. Alglave, and D. Williams. Synchronising C/C++ and POWER. In *Proc. PLDI*, 2012.

[29] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. In *Proc. PLDI*, 2011.

[30] P. Sewell, S. Sarkar, S. Owens, F. Zappa Nardelli, and M. O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *C. ACM*, 53(7):89–97, 2010.

[31] P. Sewell, F. Zappa Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strniša. Ott: Effective tool support for the working semanticist. *J. Funct. Program.*, 20(1):71–122, 2010.

[32] J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3):22:1–22:50, June 2013.

[33] D. Vytiniotis, S. L. Peyton Jones, T. Schrijvers, and M. Sulzmann. OutsideIn(X) modular type inference with local assumptions. *J. Funct. Program.*, 21(4-5):333–412, 2011.

[34] P. Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *Proc. POPL*, 1987.

[35] Freek Wiedijk. Encoding the HOL Light logic in Coq, 2007. Unpublished.

[36] J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *Proc. POPL*, 2012.