Interactive Verification of Call-by-Value Functional Programs

Arthur Charguéraud
INRIA
arthur.chargueraud@inria.fr

Abstract

A mechanized proof of total correctness enables one to verify a program with utmost confidence. Yet, setting up a methodology for reasoning formally on nontrivial code written in a general-purpose language has appeared to be a highly challenging task. In this paper, we propose a framework for modular verification of purely functional code. By embedding the syntax and semantics of a call-by-value functional language in a proof assistant, we are able to specify programs through lemmas describing their big-step behaviour, and to verify programs through proofs of such lemmas. Our framework imposes no restriction on the code, apart from its purity, and from a logical perspective is as expressive as the theorem prover being used. The practical result of this work is a technology for proving total correctness of pure Caml programs using the Coq proof assistant. We have applied our approach to fully specify and verify OCaml's list library as well as a bytecode compiler and interpreter for mini-ML.

1. Introduction

1.1 Mechanized proofs of correctness

Correctness of code is critical to ensure that programs do not crash nor compute erroneous results. To get confidence about the correctness of a program, the first step is usually to test it and proof-read its source code. To get further confidence, one may rely on static type systems and other static analysis tools, or insert runtime checks. But, in general, the only way to get full confidence in the correctness of a program is to write down a formal *specification* for it and to carry out a *mechanized proof of correctness*.

A specification is a description of what a program is intended to compute, regardless of how it computes it. Establishing correctness of a program consists in proving that this program satisfies the given specification. When the specification is written in a formal language and the proof of correctness can be verified by a machine, the proof is said to be mechanized.

Coming up with a set of definitions that allows *in theory* to mechanically prove programs correct is not such a difficult task. However, designing a system that can be used *in practice* to verify large and complex programs is much more challenging. Three aspects are particularly relevant for comparing various proposals. First, what programming and specification languages are supported? Second, how much human and machine effort is needed to complete the proofs? Third, how closely are proof scripts related to the initial source code, and are proofs easily maintainable?

1.2 Approaches to mechanized verification

Existing approaches to producing mechanically-verified software fall roughly in three categories.

Firstly, a number of tools have been implemented following the Hoare-logic approach [10]. In this style, source code is annotated with pre- and post-conditions as well as loop invariants. A *verification condition generator* (VCG) is then used to extract proof-obligations that entail the correctness of the program. These obligations are typically discharged using one or several automated theorem provers, but it is also possible to use an interactive theorem prover to carry out the proofs when necessary. This category of tools includes Caduceus [9] for verifying C programs, Spec# [2] for verifying C# programs, tools based on JML [3] for verifying Java programs (e.g. Krakatoa [14]), and Pangolin [21] which targets ML programs.

A radically different approach, called *shallow embedding*, consists in programming directly within a theorem prover and verifying properties of the code interactively inside the same framework. Indeed, the logic of a theorem prover typically contains a purely functional programming language. With this approach, a piece of code takes the form of a definition, its specification takes the form of the statement of a lemma, and its verification is carried out through the proof of that lemma. An extraction mechanism may then be used to isolate the actual source code, to be compiled into executable machine code, from proof-specific elements. In spite of a number of limitations (detailed further on), significant developments can be carried out following this approach, such as Leroy *et al*'s certified compiler [13] written and certified in Coq [5]. Several projects aim at overcoming the limitations of this approach, in particular by supporting a richer source language and providing specialized tool support. Ynot [18] extends Coq with an axiomatized monad in order to represent the impure features of the target programming language. Epigram [15] suggests programming with depend types to enforce data structures invariants, and adapting the programming language to suit dependent types. While Epigram is not strictly-speaking a shallow embedding, it can be, for our concern, considered as part of this category.

A third approach to reasoning formally on programs also relies heavily on the use of a theorem prover, but in a very different manner. It consists in describing the syntax and the semantics of a programming language in the logic of the prover. Notice that this introduces an extra level of indirection compared to directly programming in the prover. This technique is called *deep embedding* (the syntax of the programming language is embedded in the logic). Deep embedding techniques have been widely used to formally reason on properties of type systems (see [1] for a survey), to mechanize soundness and completeness proofs of logics (e.g. formalization of Hoare logics [17]), and to prove correctness of tools manipulating programs such as VCGs [11] or compilers [13].

However much less work has focused on reasoning about particular programs through a deep embedding. Existing attempts target basic procedural languages [16] or assembly-level programs [20, 7]. To the extent of our knowledge, our work is the first use of a deep embedding of a high-level programming language.

1.3 Comparison

In order to shed more light on the relative benefits and drawbacks of these three approaches, we compare them with respect to four questions: What are the constraints on the programming language? What are the constraints on the specification language? How much work is involved for verifying that a program meets its specification? And how much work is involved in order to reflect a change in the code or the specification?

Programming language A VCG (Verification Condition Generator) can be custom-made for a particular programming language. Thus, this technique does not impose any restriction on the source language. The same applies to the use of a deep embedding. Indeed, the syntax and the semantics of any given programming language can be described faithfully in a general-purpose logic.

When using a shallow embedding however, the source language needs to fit into the logical language of the theorem prover being used. In practice, this has important repercussions, most of them coming from the fact that the logical language undergoes a number of restrictions required for the sake of its soundness. In particular, partial functions need to be completed with dummy values outside their domains, exceptions have to be encoded as explicit sum types, and recursive functions must be proved terminating at the time of their definition. As a consequence, the source code needs to be largely re-engineered, and this has to be done in a clever way in order to allow for short verification proof scripts. While this re-engineering might be acceptable when writing a program from scratch, it is quite costly when starting from existing code.

Specification language The VCG approach is not commited to any particular logic, offering a lot of flexibility. A natural choice may be to rely on a standard higher-order logic. Yet, some tools choose to restrict their specification language to first-order logic, so as to simplify the task for automated theorem provers. Other tools support logics specialized for reasoning on programs, such as Separation Logic [22], which provides dedicated operators for reasoning on the contents of mutable stores.

With both the shallow and the deep embedding approaches, specifications are naturally stated in the logic of the theorem prover being used, which is presumably very expressive. Nevertheless, one may prefer to work with specifications expressed in some other logic. To that end, it suffices to define this other logic in terms of the theorem prover's logic (using either a shallow or a deep embedding of that logic).

While both a shallow and a deep embedding ultimately express specifications in terms of the theorem prover's logic, there is an important difference between the two. With a shallow embedding, the values that are being specified are those of the logic. For instance, when a shallowly-embedded program manipulates a list, it manipulates a logical list. On the contrary, with a deep embedding, one specifies *encoded* values, i.e. description of values viewed through the deep embedding. The specification can be expected to be more complex because this encoding of values has to show up somehow. One of the contributions of this work is to show that specification need not be more complex than with a shallow embedding. Intuitively, by relating encoded values with their logical counterparts, one can state specifications of encoded values at the logical level.

Verification process When using an embedding, either shallow or deep, the verification phase amounts to proving lemmas in an interactive proof environment. The set of available hypotheses and the current proof obligation are visible at each point of the program, and the user gets immediate feedback while carrying out the proof. The structure of the proof and the chaining of the main arguments that entail correctness are explicitly laid out in the script, while simple subgoals are typically discharged by invoking an external automated theorem prover.

The process is quite different when using a VCG in conjunction with automated provers. After annotating the code with its specification and loop invariants, the verification condition generator is run and the proof obligations produced are sent to one or several automated theorem provers. It sometimes happen that all the proof obligations are solved successfully, completing the job. Yet, most often, one gets back an error message reporting that the automated theorem provers have failed to prove a number of obligations. Several cases are possible: (1) the code is incorrect and does not meet the specification, (2) the code is correct but the specification or one of the invariants given does not fit the code, (3) the specification of a function being called in the code has too strong a pre-condition or too weak a post-condition, (4) the proof obligation is true but the prover simply fails to prove it. Finding out which case applies and fixing the problem is typically time-consumming, for two main reasons. First, the error usually does not come from a single fact taken individually, but rather from the failure to connect many facts together. And second, reading the details of the proof obligation that failed to be checked is long and tedious, due to the size of the proof obligation and to the fact that the intermediate results often have unfriendly machine-generated names, e.g. "`N27`".

Proof maintainability Programs are rarely written once and forall. In the case of mechanically-verified programs, a change in the source code of a program or in its specification needs to be reflected by an update in the corresponding proof of correctness. With a shallow or a deep embedding, the user can replay the proof script, which records the justification of why the program was correct before changes were made. The interactive theorem prover halts at the places where the script needs to be patched. Thus, the user has a precise view on what parts of the proof have been broken, and why they have been broken.

Once again, the process is quite different with a VCG. When running the tool after a modification of the code or its specification, the user typically gets back a new set of proof obligations that could not be discharged automatically. No explanation is provided on why the program could be verified before but no longer after the change. One particularly frustrating situation occurs when the tool fails to verify parts of the code that have *not* been modified, for the only reason that the set of hypotheses in certain proof obligations has grown just too large for the automated theorem provers to handle successfully.

Conclusion The VCG, the shallow embedding and the deep embedding approaches appear as radically different techniques to producing mechanically-verified software. Each of them comes with its relative strengths and weaknesses. Given the importance of trustworthy software, all three approaches should be worth investigating in depth. This work focuses on the deep embedding technique, which seems to have received relatively less attention in the past.

1.4 Contribution

This paper describes how one can use a deep embedding in order to specify and verify purely functional ML programs. We have deliberately considered a source language deprived of side-effects to start with. We acknowledge that programming in such a language is unrealistic for many purposes, yet our point is to show how one can handle the features of a language that are not directly related to the use of a mutable store. Support for side-effects might be added later, either through a refinement of this work, or by relying on a

fine-grained functional translation of imperative ML programs into a purely functional language (see [4]). We leave to future work the accomodation of side-effects.

We have deeply embedded the core fragment of Caml into the Coq proof assistant [5]. Our infrastructure consists of a Coq library and of an *embedder* program that translates Caml code into Coq definitions. The Coq library axiomatizes the syntax and semantics of the source language. It also defines a set of predicates for stating specifications and provides a set of reasoning rules for proving that terms admit particular behaviours. The embedder program takes as input source code written in the core fragment of Caml and generates Coq definitions that correspond to the embedding of this code, that is, to the same code described through the deep embedding.

In order to specify the behaviour of a program, the user writes a proposition that should hold of the corresponding embedded code. This proposition takes the form of a lemma stated directly in the logic of Coq, using the specification predicates defined in the library. Proving this lemma amounts to proving that the program meets its specification. Such a proof can be constructed entirely within Coq's interactive development environment. The resulting proof script can be replayed and udpated at any time, should the code or its specification be modified.

This paper makes the following contributions:

- We provide a ready-to-use methodology for specifying and verifying total correctness of call-by-value functional programs. It imposes no restriction of any kind on the source code. In particular, it supports reasoning on recursive functions, polymorphic functions, higher-order functions, first-class functions, pattern matching, exceptions, and even untyped code. Furthermore, the language of specification and proof is highly expressive since Coq is used directly.
- We introduce a family of functions called *encoders* as a way to materialize the reflection between program values and logical values. It seems that our work is the first to make use of such functions that encode logical values into an embedded language in the context of program verification. Moreover, we investigate alternative implementations of reflection and point out the benefits of using encoders.
- As far as we know, we have completed the first verification of OCaml's list library. In fact, we know of no other similar formalization for a list library based on higher-order functions and written in a realistic programming language. We also illustrate our approach with the verification of a bytecode compiler and a bytecode interpreter for mini-ML programs. Establishing the correctness of such a virtual machine involves nontrivial arguments.

In the rest of this paper, we first describe the syntax and semantics of the embedded language, focusing in particular on the definition of big-step behaviours (§2). We introduce a reflection mechanism relating values from the programming language with values from the logic, which allows us to state specifications at the level of the logic rather than directly on embedded values (§3). We then present the reasoning rules used to effectively carry out the verification of embedded programs (§4). Finally, we discuss case studies (§5), implementation (§6), related work (§7), and then conclude (§8).

2. Syntax, semantics and specification

In this section, we briefly present the embedded programming language and then introduce a few definitions used to describe big-step behaviour of programs.

2.1 Syntax of the embedded language

The programming language that we consider is a λ-calculus extended with integers, n-ary data constructors and exceptions. Moreover, functions may be recursive and perform pattern matching on their argument. The language is formally described in Figure 1. Note that for the sake of simplicity we have not included floating-points numbers and we assume integers to be arbitrarily precise.

The peculiarity of our representation of syntax is to distinguish closed values (type Val) from terms (type Trm). The decision of not including a constructor for variables in the syntax of values leads to some unfortunate duplication across the syntaxes of terms and values. Nevertheless, the resulting representation greatly simplifies the formal reasoning on substitution. Indeed, substitution is always the identity on values, by definition.

The term "$\mathsf{fix}\, x_f\, p_1\, t_1\, t_2$" describes a function named x_f with input pattern p_1. Its subterm t_1 describes the continuation to be followed when the argument provided to the function matches p_1, and t_2 describes the contination to be applied otherwise. We define a conventional abstraction "$\mathsf{abs}\, x\, t$" as "$\mathsf{fix}\ \_\ (\mathsf{pvar}\, x)\, t\ \_$". Throughout the paper, we let the meta-variable f range over values built upon the constructor vfix, and we write "$\mathsf{App}\, f\, v$" as a shorthand for "$\mathsf{app}\ (\mathsf{val}\, f)\ (\mathsf{val}\, v)$".

2.2 Semantics and specification of terms

The language is equipped with a deterministic, call-by-value semantics. Determinism is not essential, it simply allows for a few simplifications. Preliminary experiments suggest that the material presented in this paper can be generalized so as to support a non-deterministic language. The restriction to call-by-value semantics is however important. Reasoning on termination and exceptions under other reduction strategies (e.g. call-by-name) is theoretically feasible, but seems to be significantly harder.

The semantics of the source language is defined using a small-step reduction relation. The predicate $t \longrightarrow t'$ indicates that the term t reduces in one step towards t'. The details of this reduction relation are not required to understand the rest of this paper. Its definition can be found in our Coq development.[1]

While small-step semantics is a convenient way of defining reductions, reasoning on programs is more naturally carried out in big-step style. In a deterministic call-by-value semantics, a given term admits one of the following big-step behaviours:

- **Returns a value** A term t evaluates to a value v if it reduces in a finite number of steps to the value v. This is formally written "$t \longrightarrow^* \mathsf{val}\, v$", where the relation $\longrightarrow^*$ is the reflexive-transitive closure of the reduction relation $\longrightarrow$. In practice, it is often more convenient to state that the term t converges to *some* value that satisfies a given predicate P, without necessarily being explicit about which value it converges to. We write this as "$t \rhd \mid P$", which reads "t returns a value satisfying P".
- **Throws an exception** A term t throws an exception v if it reduces to an exception, i.e. "$t \longrightarrow^* \mathsf{exn}\, v$". The value v describes the exception and may be analysed by exception handlers. Remark that v is often a constant data constructor, e.g. $\mathsf{NotFound}$. The notation for terms throwing exceptions is "$t \rhd! v$", which reads "t throws the exception v".
- **Diverges** A term t diverges if an infinite sequence of reduction steps begins on t, which we write "$t \rhd \Uparrow$". This is equivalent to saying that all finite reduction sequences starting on t must end with a term that is further reducible:

$$\mathsf{diverges}\, t \quad \equiv \quad \forall t'.\ (t \longrightarrow^* t') \ \Rightarrow\ \exists t''.\ (t' \longrightarrow t'')$$

[1] The formal development associated with this paper can be found at: `http://arthur.chargueraud.org/research/2009/deep/`.

Variable	(type Var)	x		(variables are implemented using natural numbers)
Constructor	(type Con)	c		(data constructors are implemented using natural numbers)
Terms	(type Trm)	t	:=	$\mathsf{val}\, v \mid \mathsf{var}\, x \mid \mathsf{app}\, t_1\, t_2 \mid \mathsf{constr}_n\, c\, t_1 \ldots t_n \mid \mathsf{fix}\, x_f\, p_1\, t_1\, t_2 \mid \mathsf{exn}\, v \mid \mathsf{try}\, t_1\, \mathsf{with}\, t_2$
Values	(type Val)	v	:=	$\mathsf{vint}\, i \mid \mathsf{vbuiltin}\, b \mid \mathsf{vconstr}_n\, c\, v_1 \ldots v_n \mid \mathsf{vfix}\, x_f\, p\, t_1\, t_2$
Patterns	(type Pat)	p	:=	$\mathsf{pvar}\, x \mid \mathsf{pint}\, i \mid \mathsf{pconstr}_n\, c\, p_1 \ldots p_n \mid \mathsf{pany} \mid \mathsf{palias}\, x\, p$
Primitive	(type Prm)	b	:=	$\mathsf{add} \mid \mathsf{sub} \mid \mathsf{mult} \mid \mathsf{div} \mid \mathsf{cmp} \mid \mathsf{leq} \mid \mathsf{throw}$

Figure 1. Embedding of the syntax

- **Gets stuck** A term t is stuck if it is irreducible and neither a value nor an exception. In practice, we do not care much about specifying that programs get stuck. Instead, we introduce a more general "unspecified" behaviour. The proposition "$t \rhd ?$" indicates that the behaviour of t is not specified, and it holds of any term t.

Formally, we introduce a general binary relation between terms and behaviours. The predicate "$t \rhd B$" indicates that the term t admits the behaviour B. The grammar of behaviours is:

$$B \quad := \quad (|\, P) \quad | \quad !\, v \quad | \quad \Uparrow \quad | \quad ?$$

The relation $\rhd$ is defined using one introduction rule per behaviour:

$$\text{RETURNS-VAL}\ \frac{P\, v \qquad t \longrightarrow^* \mathsf{val}\, v}{t \rhd (|\, P)} \qquad \text{THROWS}\ \frac{t \longrightarrow^* \mathsf{exn}\, v}{t \rhd (!\, v)} \qquad \text{DIVERGES}\ \frac{\mathsf{diverges}\, t}{t \rhd (\Uparrow)} \qquad \text{UNSPEC}\ \frac{}{t \rhd (?)}$$

2.3 Specification of functions

In the Hoare-style methodology [10], a function is specified using a pre-condition and a post-condition. The pre-condition is a sufficient condition on the argument of a function for ensuring that the application of the function to its argument executes safely. The post-condition is a binary predicate that relates the output of the function to its input, in case the application of the function terminates. When total correctness is aimed at, termination is proved through an analysis of loop structures and recursive function calls.

With a deep embedding, we can define a single judgment that captures the fact that a function, when applied to an argument satisfying its pre-condition, executes safely, terminates, and returns a value satisfying its post-condition. The following proposition states that for any input value v that satisfies the pre-condition P, the function f terminates without error and returns a value v' such that the post-condition Q holds of v and v'.

$$\forall\, v. \quad (P\, v) \quad \Rightarrow \quad \exists\, v'.\ (\mathsf{App}\, f\, v) \longrightarrow^* (\mathsf{val}\, v') \ \wedge\ (Q\, v\, v')$$

Using the return predicate that we have introduced earlier, we may restate the same proposition as:

$$\forall\, v. \quad (P\, v) \quad \Rightarrow \quad (\mathsf{App}\, f\, v) \ \rhd\ |\, (Q\, v)$$

It is sometimes the case that both the pre- and the post-condition need to be expressed in terms of some logical variables, the so-called *auxiliary variables* in Hoare logic. (Illustration of auxiliary variables appears in the case studies section, §5.) To allow for the universal quantification of such variables in the previous statement, we consider a more general form of specification for functions. In our setting, the specification of a function f is just an arbitrary proposition K that relates an argument v to the behaviour of the application of f to v. The predicate K has the type "$\mathsf{Val} \rightarrow \mathsf{Trm} \rightarrow \mathsf{Prop}$", where Prop is the type of logical propositions. The corresponding specification predicate is defined as follows:

$$\mathsf{spec}\, f\, K \quad \equiv \quad \forall\, v.\ K\, v\, (\mathsf{App}\, f\, v)$$

Examples of use appear further on the case-studies section (§5).

The behaviour of curried n-ary functions can be described in a similar fashion. For instance, predicate spec_2 is used to specify a function of two arguments as a function that, when applied to a first argument, returns a unary function verifying an instance of the predicate spec. In the formal definition of spec_2 which follows, predicate K has type "$\mathsf{Val} \rightarrow \mathsf{Val} \rightarrow \mathsf{Trm} \rightarrow \mathsf{Prop}$".

$$\mathsf{spec}_2\, f\, K \quad \equiv \quad \mathsf{spec}\, f\, (\lambda v. \lambda t.\ t \ \rhd\ |\, (\lambda g.\ \mathsf{spec}\, g\, (K\, v)))$$

In summary, we have presented a set of definitions which can be used to state formally that a certain term admits a certain behaviour and that a certain function admits a certain specification. The definitions of the return behaviour as well as the predicate spec will be generalized in the next section so as to take reflection into account.

3. Reflection of values into the logic

Reflection is a mechanism that relates values of the programming language to their counterpart in the logic, whenever possible. This mechanism plays a central role in our framework: without it, all specifications and proofs would be polluted with details of the deep embedding. The matter of this section is to show how we implement reflection using functions called *encoders*, and to explain how we use encoders in specifications. Moreover, we will investigate other possibilities for the implementation of reflection and argue why encoders appear to be the most appropriate choice for writing specifications.

3.1 Introduction to encoders

Through the deep embedding, we can reason on untyped programs. Yet, in order to write specification and carry out proofs in terms of logical values (e.g. Coq's lists) rather than in terms of values described in the embedded syntax, we exploit the fact that most often the data-structures manipulated by a program are in fact well-typed in ML. The key observation is that for each well-typed value from the embedded language there exists a corresponding logical value. For instance, consider the program value "4 :: nil", which admits the ML type "int list". This object is described in our embedding by a value of type Val, namely:

$$\mathsf{vconstr}_2\ \mathsf{cons}\ (\mathsf{vint}\ 4)\ (\mathsf{vconstr}_0\ \mathsf{nil})$$

At the same time, this object corresponds to the Coq list "4 :: Nil", of logical type "List Int".

The purpose of this section is to formally establish this relationship. First, we explain how to define, for each ML type, the Coq type that corresponds to it. Second, we set up for each ML type a logical function that connects embedded values of this type with their logical counterparts. Remark: in the following, we do not consider the case of recursive types. To start with, we assume that the data-structures are free of first-class functions, in other words that their types do not contain arrow constructor.

Given an ML algebraic type declaration or a type abbreviation, we create the corresponding type definition in Coq. This translation is just a matter of adapting the syntax. We illustrate this process on booleans, polymorphic lists, and lists of booleans, giving

the ML type definitions, followed with the corresponding logical definitions. Caml syntax and Coq syntax are used, with the exception that ML constants are written in lowercase and Coq constants are written in uppercase, for the sake of presentation.

```
>  type bool = true | false
>  type 'a list = nil | cons of 'a * 'a list
>  type bitlist = bool list
Inductive Bool : Type :=
   | True : Bool
   | False : Bool.
Inductive List (A:Type) : Type :=
   | Nil : List A
   | Cons : A -> List A -> List A.
Definition Bitlist := List Bool.
```

Our second step is to relate values of the embedded language with their logical equivalent. To that end, we use functions that translate logical values into values of the embedded language. Because these functions encode logical values using the constructs of the deep embedding, we call them *encoders*. For example, the encoder for booleans is a logical function that takes logical values, of type Bool, and produces embedded values, of type Val. We name this encoder _Bool. In the definition which follows, the lowercase constants true and false are distinct constants representing data constructors of the embedded language (type Con), while the uppercase constants True and False describe the booleans from the logic (type Bool).

```
Definition _Bool (b:Bool) : Val :=
 match b with
 | True  => vconstr_0 true
 | False => vconstr_0 false
 end.
```

Slightly more complex is the encoder for lists, named _List. If A is a logical type and l is a logical list of type List A, and if _A is an encoder for values of type A, then "_List A _A l" computes the embedded value that corresponds to l. In other words, the term _List is a polymorphic encoder such that, for any type A with its associated encoder _A, "_List A _A" translates lists of type List A into values of type Val. This encoder is defined as follows:

```
Fixpoint _List (A:Type)(_A:A->Val)(l:List A):Val :=
 match l with
 | Nil      => vconstr_0 nil
 | Cons h t => vconstr_2 cons (_A h) (_List A _A t)
 end.
```

Finally, we define the encoder for lists of booleans by specializing the encoder of polymorphic lists to the type of booleans:

```
Definition _Bitlist : List Bool -> Val :=
  _List Bool _Bool.
```

More generally, for any ML type, we define the corresponding type A in the logic, and define the associated encoder of type "A → Val". This encoder is, by convention, named _A. It is such that for any logical value X of type A, the application "$\_A\,X$" is the representation in the deep embedding of the program value that corresponds to X.

This reflection mechanism applies to data-structures, but not to functions. We do not reflect program functions as logical functions. Indeed, while a particular program function may be shown to implement a logical function on a given domain, the set of all program functions of a given type is not in bijection with the set of logical functions of the corresponding type. To specify an embedded function, that is, a value of type Val built with the constructor vfix, we simply give a predicate of type "Val → Prop" that should hold of this function. It is typically an instance of the predicate spec.

In ML, since functions are first-class values, one may work with, say, a list of functions on integers. Such a list admits the ML type "(int → int) list", and we would like to be able to reflect this value into the logic as a logical value of type "List Val". In order to be able to reuse the encoder _List for polymorphic lists, we define the function _Val as the identity function on the type Val. Thereafter, "_List Val _Val" is the encoder for lists of any kind of value, including lists of functions. More generally, our translation from ML types to their logical equivalent maps all arrow types to the logical type Val.

Given an ML type, the generation of the corresponding type declaration and of the associated encoder is entirely mechanical. We have implemented a program which, given a set of type declarations written in Caml syntax, generates all the corresponding Coq definitions.[2] This program saves the user the trouble of writing these tedious definitions by hand.

3.2 Specification using encoders

We now explain how the reflection mechanism is exploited to specify return values of terms and input values of functions.

First, we integrate encoders with the behaviour "returns". The predicate "$t \rhd \_A \mid P$" indicates that the term t returns the translation by the encoder _A of a logical value of type A that satisfies the predicate P. The predicate P is now a predicate of type "A → Prop" rather than "Val → Prop". We are thereby lifting specifications from the level of embedded values to the level of logical values. We generalize the rule RETURNS-VAL accordingly, stating that t returns the encoding "$\_A\,V$" of a logical value V of type A that satisfies P.

$$\frac{t \longrightarrow^* \mathsf{val}\,(\_\mathsf{A}\,V) \qquad P\,V}{t \rhd \_\mathsf{A} \mid P}\ \text{RETURNS}$$

Note that this new definition is equivalent in terms of expressiveness to the earlier one: "$t \rhd \mid P$" is equivalent to "$t \rhd \_\mathsf{Val} \mid P$", and "$t \rhd \_A \mid P$" is equivalent to "$t \rhd \mid (\lambda v.\, \exists V.\, v = \_\mathsf{A}\,V \wedge P\,V)$".

Dually, the reflection mechanism is exploited in the specification of function arguments. The predicate "Spec f A _A K" describes the behaviour of the application of f to the encoding by $\_A$ of a logical value V, in terms of V. More precisely, K is a binary predicate that relates V to the application of f to "$\_A\,V$". The formal definition of Spec is:

$$\mathsf{Spec}\,(f : \mathsf{Val})\,(A : \mathsf{Type})\,(\_A : A \to \mathsf{Val})\,(K : A \to \mathsf{Trm} \to \mathsf{Prop})$$
$$\equiv \quad \forall\,(V : A).\ K\,V\,(\mathsf{App}\,f\,(\_A\,V))$$

For example, consider the function neg which returns the opposite of a given integer. It is such that for any integer n, the term $(\mathsf{App}\,\mathsf{neg}\,(\_\mathsf{Int}\,n))$ reduces to the value $(\_\mathsf{Int}\,(-n))$. Its specification is:

$$\mathsf{Spec}\ \ \mathsf{neg}\ \ \mathsf{Int\_Int}\ \ (\lambda n.\,\lambda t.\,(t \rhd \_\mathsf{Int} \mid = -n))$$

Unfolding the definition of Spec, this specification is equivalent to:

$$\forall\,n.\,\exists\,m.\ \ (\mathsf{App}\,\mathsf{neg}\,(\_\mathsf{Int}\,n)) \longrightarrow^* (\_\mathsf{Int}\,m) \ \ \wedge \ \ m = -n$$

Once again, the use of encoders allows to state propositions in terms of logical values (here n and m) rather than in terms of the corresponding embedded values. Notice that the behaviour of the function neg is unspecified on values which are not the translation by the encoder _Int of a logical value of type Int. In other words, neg is simply not specified when its argument is not a value of the form "vint n".

[2] For common types, such as bool or list, we reuse the definitions from Coq's standard library rather than generating new definitions.

- **Working with a relation**

$$\begin{array}{l}\forall\,(l : \mathsf{Val}).\,\forall\,(L : \mathsf{List}\,A).\ (R\,L\,l)\\ \Rightarrow\ \exists\,(n : \mathsf{Val}).\,\exists\,(N : \mathrm{Int}).\ (\mathsf{App}\,\mathsf{length}\,l \longrightarrow^{*} n)\\ \qquad\qquad\qquad\qquad \wedge\ (R\,N\,n)\\ \qquad\qquad\qquad\qquad \wedge\ (\mathsf{Coq.length}\,L = N)\end{array}$$

- **Working with decoders**

$$\begin{array}{l}\forall\,(l : \mathsf{Val}).\,\forall\,(L : \mathsf{List}\,A).\ (\mathsf{Some}\,L = \bar{}\mathsf{List}\,A\,\bar{}A\,l)\\ \Rightarrow\ \exists\,(n : \mathsf{Val}).\,\exists\,(N : \mathrm{Int}).\ (\mathsf{App}\,\mathsf{length}\,l \longrightarrow^{*} n)\\ \qquad\qquad\qquad\qquad \wedge\ (\mathsf{Some}\,N = \bar{}\mathrm{Int}\,n)\\ \qquad\qquad\qquad\qquad \wedge\ (\mathsf{Coq.length}\,L = N)\end{array}$$

- **Working with encoders**

$$\begin{array}{l}\forall\,(L : \mathsf{List}\,A).\,\exists\,(N : \mathrm{Int}).\\ \qquad (\mathsf{App}\,\mathsf{length}\,(\_\mathsf{List}\,A\,\_A\,L)) \longrightarrow^{*} (\_\mathrm{Int}\,N)\\ \qquad \wedge\ (\mathsf{Coq.length}\,L = N)\end{array}$$

Figure 2. A side-by-side comparison of three approaches to implementing reflection, on a function computing the length of a list

3.3 Other implementations of reflection

To conclude this section, we investigate how can one implement, in the logic, the relationship between embedded values and logical values. Encoders are one possibility, but certainly not the only one.

A naive attempt is to define reflection using a binary relation. Since embedded values have type Val and logical values have a type A that depends on the program value they correspond to, reflection can be implemented using a dependently typed predicate, call it R, of type "$\forall\,A,\ A \to \mathsf{Val} \to \mathsf{Prop}$". For instance, "$R\,\mathsf{Bool}\,\mathsf{True}\,(\mathsf{val\_constr}_0\,\mathsf{true})$" describes the fact that the program boolean value true is mapped onto the logical boolean value True. Similarly, "$R\,(\mathsf{List\ A})\,\mathsf{Nil}\,(\mathsf{val\_constr}_0\,\mathsf{nil})$" describes the reflection of constructor nil as the empty list of type A, and it holds for any A. Working directly with relation R is possible, as demonstrated for instance by Mehta and Nipkow [16], yet it is quite heavy (an example appears further on). So, instead, we try and define R as a function.

The example of the empty list shows that a given untyped value may correspond to several logical values, such as empty lists of different types. Thus, relation R cannot be implemented as a function from program values to logical values. However, given a type A, a given program value is reflected by at most one logical value of type A. Thus, we may consider a family of functions: for each logical type A that corresponds to a programming language type, we can define a function $\bar{}A$ of type "$\mathsf{Val} \to \mathsf{Option}\,A$" able to decode values that correspond to a logical value of type A (it returns None for other values). We call these functions *decoders*.

Symmetrically, we may try to implement R using a function from logical values towards program values. It should be feasible since each logical value is mapped to at most one program value. The corresponding function would have type "$\forall\,A, A \to \mathsf{Val}$". Yet, it is impossible to define in the logic a function of this type with the intended behaviour, because one cannot perform a case analysis on a polymorphic argument (due to parametricity of the logic). The simplest solution to work around this issue is, again, to implement relation R using not one but a family of functions. More precisely, for each logical type A which corresponds to a programming language type, we define a function of type "$A \to \mathsf{Val}$" which maps each logical value of type A to its corresponding program value. These functions are the ones which we call *encoders*.

We now compare the three possible approaches: directly working with the reflection relation R, working with decoders, or working with encoders. Figure 2 shows how these three approaches apply to a simple yet informative example: the specification of a function which computes the length of a list in terms of the logical list length function (written Coq.length). While the first two approaches are roughly equivalent, the third one based on encoders has two significant advantages.

First, it involves a conjunction of only two propositions, while the other approaches require four propositions. This comes from the fact that the application of encoders can be inlined in the proposition stating the reduction relationship. Second, only variables at the logical level need to be quantified when using encoders. In other words, there is no need to quantify over variables describing encoded terms (i.e. variables of type Val). Again, this is a significant gain when stating specifications. Thus, while it is undoubtedly possible to reason on programs without encoders, the use of encoders appears to be the most effective way of specifying embedded programs at the logical level.

3.4 Inference of specifications using decoders

While we use exclusively encoders in the specification of programs, we actually rely on the help of decoders during the verification phase. Roughly speaking, we use decoders as a way to shorten proof scripts. To understand how decoders help us, consider the following piece of code which describes a list of length one containing the value 4.

$$\mathsf{val}\ (\mathsf{vconstr}_2\ \mathsf{cons}\ (\mathsf{vint}\ 4)\ (\mathsf{vconstr}_0\ \mathsf{nil}))$$

During the verification of a program which contains this code, it is possible to state explicitly that the behaviour of this subterm is "$\_\mathsf{List}\ \_\mathrm{Int} \mid\ =\ 4 :: \mathsf{Nil}$". But in fact, this specification can be automatically inferred from the code, using decoders. This saves the user from paraphrasing a large amount of the source code of the program being verified.

To avoid difficulties associated with the type-checking of decoders, we implement decoding as a function written in the language of tactics of Coq. Contrary to logical functions, tactics are untyped and may fail. In particular, decoding only succeeds for a program value if it corresponds to some logical value whose type is not Val.

4. Reasoning on embedded programs

In this section, we explain how to prove that a given piece of code admits a given behaviour. Such proofs are carried out using a set of reasoning rules, all of which are formally proved correct with respect to the semantics of the embedded language. Deriving reasoning rules from the underlying semantics is not a novel idea. It seems to have been pioneered by M. Gordon over two decades ago [17]. Our contribution here lies in the design of a particular set of reasoning rules that makes verification of ML programs both intuitive and practical.

4.1 Case study: a rule for let-expressions

To introduce to the underlying mechanisms involved in our reasoning rules, we construct a reasoning rule for "let" expressions step by step. Remark: "$\mathsf{let}\,x\ =\ t_1\,\mathsf{in}\,t_2$" is encoded in our embedded language as "$\mathsf{app}\,(\mathsf{abs}\,x\,t_2)\,t_1$".

Step 1 If a term t reduces to a term t', then t and t' admits exactly the same behaviours. Formally:

$$t \longrightarrow t' \quad \Rightarrow \quad (t \rhd B \iff t' \rhd B)$$

Thus, to prove that a term t admits a behaviour B, it suffices to show that t reduces to a term t' such that t' admits the behaviour B (this is formally stated by the rule REDUCTION in Figure 3).

In particular, to show that "$\mathsf{let}\,x\ =\ t_1\,\mathsf{in}\,t_2$" admits the behaviour B it suffices to show that t_1 evaluates to some value v_1 and that "$[x \to \mathsf{val}\,v_1]\,t_2$" admits the behaviour B. For improved

$$\textsc{reduction}\ \frac{t \longrightarrow^* t' \qquad t' \rhd B}{t \rhd B} \qquad \textsc{ctx-returns}\ \frac{t \rhd \_A \,|\, P \qquad \mathsf{redctx}\, C \qquad \forall X.\,(P\,X) \;\Rightarrow\; C[\mathsf{val}\,(\_A\,X)] \rhd B}{C[t] \rhd B}$$

$$\textsc{ctx-throws}\ \frac{t \rhd !\, v \qquad \mathsf{redctx}\, C \qquad C[\mathsf{exn}\, v] \rhd B}{C[t] \rhd B} \qquad \textsc{ctx-diverges}\ \frac{t \rhd \Uparrow \qquad \mathsf{redctx}\, C}{C[t] \rhd \Uparrow}$$

$$\textsc{spec-intro}\ \frac{\forall X.\,(K\,X\,(\mathsf{App}\,f\,(\_A\,X)))}{\mathsf{Spec}\,f\,A\,\_A\,K} \qquad \textsc{spec-elim}\ \frac{\mathsf{Spec}\,f\,A\,\_A\,K}{K\,V\,(\mathsf{App}\,f\,(\_A\,V))}$$

$$\textsc{spec-weaken}\ \frac{\mathsf{Spec}\,f\,A\,\_A\,K' \qquad \forall X\,t.\;(K'\,X\,t) \;\Rightarrow\; (K\,X\,t)}{\mathsf{Spec}\,f\,A\,\_A\,K} \qquad \textsc{spec-induction}\ \frac{\mathsf{Spec}\,f\,A\,\_A\,(\lambda X t.\; H \;\Rightarrow\; K\,X\,t) \quad \text{where} \quad H \equiv \mathsf{Spec}\,f\,A\,\_A\,(\lambda X' t'.\; X' \prec X \;\Rightarrow\; K\,X'\,t')}{\mathsf{Spec}\,f\,A\,\_A\,(\lambda X t.\; K\,X\,t)}$$

Figure 3. Reasoning rules

readibility, we give the formal statement below under the form of an inference rule, although it corresponds to a proven theorem.

$$\frac{t_1 \longrightarrow^* \mathsf{val}\, v_1 \qquad ([x \to \mathsf{val}\, v_1]\, t_2) \rhd B}{(\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2) \rhd B}$$

Reasoning on a program using such a symbolic evaluation is not practical. Indeed, it requires that, for each term, we specify to which exact value it evaluates. In general we wish to be more abstract, and only express that a given property holds of the result of its evaluation. Furthermore, performing the naive substitution of v_1 into t_2 may lead to a significant increase in the size of the proof obligation.

Step 2 Let P be the property that we wish to hold of the result v_1 of the evaluation of t_1. To show that "$\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2$" admits the behaviour B, it suffices to show that t_2 admits the behaviour B under the assumption that the property P holds of x, that is "$P\,x$". An intuitive —although technically ill-formed— statement for reasoning on a let expression could be:

$$\frac{t_1 \longrightarrow^* \mathsf{val}\, v_1 \qquad P\, v_1 \qquad \forall x.\,\big(P\,x \;\Rightarrow\; t_2 \rhd B\big)}{(\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2) \rhd B}$$

The above rule is ill-formed because x is used both as a program variable in the conclusion and as a meta-variable of type Val in the premise. The correct version, shown below, assigns the name X to the meta-variable, and performs the substitution of the program variable x (technically "$\mathsf{var}\, x$") with the value X in t_2.

$$\frac{t_1 \longrightarrow^* \mathsf{val}\, v_1 \qquad P\, v_1 \qquad \forall X.\;(P\,X) \;\Rightarrow\; ([x \to \mathsf{val}\, X]\, t_2) \rhd B}{(\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2) \rhd B}$$

Step 3 Finally, this latter rule needs to be slightly extended to account for the use of encoders. So, instead of letting X stand for the program value to which the term t_1 evaluates, we let X stand for the corresponding logical value. Thus, if A is the logical type of the result of t_1 and if $\_\mathsf{A}$ is the encoder for this type, then the result of the evaluation of t_1 is described by the value "$\_\mathsf{A}\,X$".

Thereafter, to prove that "$\mathsf{let}\, x \;=\; t_1\, \mathsf{in}\, t_2$" admits a behaviour B, it suffices to show that t_1 returns the encoding by $\_\mathsf{A}$ of a value satisfying P, and that "$[x \to \mathsf{val}\,(\_\mathsf{A}\,X)]\, t_2$" admits the behaviour B under the assumption that "$P\,X$" holds. The corresponding reasoning rule is stated below.

$$\frac{t_1 \rhd \_A \,|\, P \qquad \forall X.\;(P\,X) \;\Rightarrow\; ([x \to \mathsf{val}\,(\_A\,X)]\, t_2) \rhd B}{(\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2) \rhd B}$$

Summary Starting from a statement similar to a big-step evaluation rule, we have introduced a meta-variable X to name the logical representation of the result of the intermediate computation, and we have then introduced a predicate P describing the post-condition that holds of this result. By doing so, we have obtained a general and useful rule for reasoning on let-expressions. This rule can be easily generalized to other reduction contexts, as explained next.

4.2 Reasoning rules

The set of reasoning rules used in our framework is presented in Figure 3 and described below. Note that these rules, which are actually theorems, are provided only to help carrying out proofs of correctness; expressiveness is not constrained by these rules since one is not forced to use them.

Reduction in context We start with a definition of reduction contexts. A meta-level function C, of type "$\mathsf{Trm} \to \mathsf{Trm}$", is a reduction context if whenever a term t takes a reduction step the application of C to t, written $C[t]$, reduces accordingly. Formally:

$$\mathsf{redctx}\, C \quad \equiv \quad \forall t\, t'.\;(t \longrightarrow t') \;\Rightarrow\; (C[t] \longrightarrow C[t'])$$

When C is a reduction context, the behaviour of a term $C[t]$ depends on the behaviour of t.

- If t returns a value, say "$t \rhd \_\mathsf{A} \,|\, P$", then, to prove that $C[t]$ admits a behaviour B, it suffices to prove that for any logical value X of type A such that $(P\,X)$ holds, the term $C[\mathsf{val}\,(\_\mathsf{A}\,X)]$ admits the behaviour B. This is stated formally by rule CTX-RETURNS in Figure 3.
- If t throws an exception v, then the behaviour of $C[t]$ is the same as the behaviour of $C[\mathsf{exn}\, v]$. (See rule CTX-THROWS.) To reason on this later term, one typically reduces the term using rule REDUCTION so as to propagate the exception until reaching either an exception-handler or the root of the term.
- If t diverges, then $C[t]$ diverges as well (rule CTX-DIVERGES).

Applications Consider a term t in beta-redex form. It is of the form "$\mathsf{App}\,f\,(\_\mathsf{A}V)$", where f is a closed value describing the function and V is the logical value describing the argument. (If the argument has no logical equivalent, then V is of type Val and $\_A$ is the encoder $\_\mathsf{Val}$ defined as the identity function in §3.1.) To reason on the term t, there are two possible situations, depending on whether function f has already been specified or not.

- If a specification for f is available, say "$\mathsf{Spec}\,f\,A\,\_A\,K$", then the proposition "$K\,V\,t$" describes the behaviour of t. This comes directly from the definition of Spec, and is implemented by the rule SPEC-ELIM.
- However, there might not yet exist a specification for the function f. This happens in particular when we are in the process of proving a specification for the function f. In this case, we beta-reduce the application of f to its argument in order to reason on the application. In the particular case where f is a simple abstraction, say "$\mathsf{abs}\, x\, t_1$", the reasoning rule that corresponds to the contraction of the beta-redex is:

$$\textsc{reduction-beta}\ \frac{[x \to \mathsf{val}\,(\_A\,V)]\, t_1 \rhd B}{\mathsf{App}\,(\mathsf{abs}\, x\, t_1)\,(\_A\,V) \rhd B}$$

The above rule is in fact an instance of the more general rule REDUCTION from Figure 3, which allows to execute embedded programs step by step. The rule REDUCTION is used to reason on beta-reduction and pattern-matching, as well as propagation of exceptions and exception-catching.

Functions If the term studied is a function f (that is, a value built using the constructor vfix), then we can prove a specification for it of the form "Spec $f\ A\ \_A\ K$" in one of two ways:

- Either we apply the definition of Spec, that is, we show that for any argument X the proposition "$K\ X\ t$" holds, where t is the application of f onto "$\_\mathsf{A}\ X$". This is implemented by the rule named SPEC-INTRO.
- Or we show that this specification weakens another one proved previously, say "Spec $f\ A\ \_A\ K'$", by proving that for any X and t, the proposition "$K'\ X\ t$" implies the proposition "$K\ X\ t$". This is formalized in the rule SPEC-WEAKEN.

Proving the correctness of a recursive function generally involves reasoning by induction. To that end, one can either use the primitive induction tactic of Coq or use directly our high-level reasoning rule SPEC-INDUCTION. With this rule, one can prove a given specification for f by verifying only a weaker specification for f. This weaker specification allows to assume that the specification which we are trying to prove already holds of the function f for any call on a strictly-smaller argument. In the rule SPEC-INDUCTION, this induction hypothesis is written H, and the symbol $\prec$ denotes a well-founded order which is to be provided by the user.

4.3 Reasoning tactics

The reasoning rule presented in the previous section can be applied in an interactive proof as any other lemma, using Coq's tactic named "apply". Nevertheless, we introduce a set of tactics to help applying these rules more conveniently. More precisely, the role of these tactics is to select the reasoning rule that applies, help instantiate its premises, and prove trivial subgoals. For instance, tactics are able to find out what the current reduction context is, apply the corresponding context reasoning rule, and discharge side-conditions such as "redctx C".

Three techniques are used to help with instantiation of premises. First, tactics may guess instantiations from information available in the proof context. Secondly, they may introduce existential variables for delaying the instantiation until more information becomes available. Thirdly, tactics rely on a database of lemmas that enables them to automatically come up with the name of the specification lemma for a given function, provided there exists one. Thereby, the behaviour of the application of a known function can be deduced automatically. The use of tactics is illustrated in the next section.

5. Case studies

In theory, a deep embedding of a programming language in a sufficiently-expressive theorem prover can be used to prove any true property of any given program. In practice, however, only a well-designed and carefully-optimized framework will be successful at verifying total correctness of realistic programs. Two properties are crucial for success.

- First, specifications should be succinct and intuitive, so that they may be proof-read by third-party individuals, even those unfamiliar with formal proofs. Indeed, specifications of functions exposed in the interface are part of the trusted base.
- Second, the technology should be such that proof scripts have a length linear in the size and in the complexity of the code being verified. Moreover, the constant factor involved should be reasonably small. This is a necessity for the technology to have a chance to accommodate large programs.

```
let merge_sort cmp l =
  let rec sort n l =
    match n, l with
    | 1, x::_ -> [x]
    | _ ->
       let k = n / 2 in
       merge cmp (sort k l) (sort (n-k) (drop k l))
    in
  let len = length l in
  if len <= 1 then l else sort len l
```

Figure 4. Source code of a merge sort function

The purpose of the case studies that we have carried out is to argue that our framework meets these two requirements on pieces of typical functional code.

5.1 Notation for specifications

Before presenting the case studies, we describe briefly the plain text notation used in specifications. The syntax for behaviours matches the notation used in this paper. For instance, "`t |> B`" corresponds to "$t \triangleright B$", and "`t |> _Int | = 3`" states that the term t returns the encoding of an integer equal to 3. For brevity, we exploit Coq's implicit arguments feature, which allows to write "`_List _Int`" to mean "`_List Int _Int`".

For specifications, we use Coq's notation feature to introduce the following syntax:

```
spec f [x:_A] = t is K
```

which reads: "the application of the function f to an argument x is a term t whose behaviour is described by K". It is defined as "Spec $f\ A\ \_A\ (\lambda x.\, \lambda t.\, K)$", where both x and t are bound in K. Note that the type A is deduced from the type of the encoder $\_A$. Indeed, $\_A$ must have the type "$A \to \mathsf{Val}$".[3]

Finally, in the particular case where the function does not require any pre-condition nor any auxiliary variable, we write

```
spec f [x:_A] is B
```

as short for "`spec f [x:_A] = t is t |> B`".

5.2 Example: merge sort

We start with a detailed example: the specification and verification of a merge sort function. This function, whose source code appears in Figure 4, is a simplified version of the sorting function implemented in OCaml's list library.

Its implementation relies on a recursive function named sort, which takes two arguments. Intuitively, "sort $n\ l$" sorts the first n items from the list l. The use of the parameter n avoids the need for constructing new lists when making recursive calls, and thus leads to improved performance. The implementation of merge_sort relies on three auxiliary functions: "length l" computes the length of a list l, "drop $k\ l$" returns the sublist obtained by removing the k first elements of l, and "merge $l_1\ l_2$" returns a sorted list that contains the union of the elements of l_1 and l_2, provided l_1 and l_2 are themselves sorted. Note that we do not require the code of these three functions in order to verify merge_sort, but only their specification.

The specification of the function merge_sort is given by the statement of the lemma that appears at the top of Figure 5. This

[3] We are looking forward to exploit the new type-class mechanism of Coq in order to deduce the parameter $\_A$ from the type of K in "Spec $f\ A\ \_A\ K$". We should then be able to simply write: "`spec f x = t is K`".

```
Lemma merge_sort_spec :
  forall (A : Type) (_A : A -> Val)
   (le : relation A) (Le : total_pre_order.rel le),
  spec merge_sort [cmp:_Val] [l:_List _A] = t is
    comparator cmp _A le ->
    t |> _List _A | sorts le l.
Proof.
intros. xintros cmp l. introv Cmp. xred.
(* -- reasoning on sort -- *)
xinfun (fun f =>
 spec f [n:_Int] [l:_List _A] = t is
 n >= 1 -> n <= length l ->
 t |> _List _A | sorts le (truncate (abs n) l))
 as sort.
clear l. xinduction (unproj21 (list A) (downto 0)).
xintros n l. introv sort_spec Nge2 Nlel. xreds 3.
asserts [[x [q [Neq1 L]]]|Nneq1]:
 ((exist x q, n = 1 /\ l = x::q) \/ n != 1).
  testsb Neq1: (n==1).
    left. destruct l as [|x q]; tryfalse. exists~.
    right~.
(* -- -- case n = 1 -- *)
subst n l. xpat. xreds. xreturns~.
(* -- -- case n > 1 -- *)
xredfail. simpl. rewriteb~ Nneq1. xreds.
xapply~ as n1. xred.
forwards~ Q1 L1 Q2 L2: (@div_2_parts' n n1 (n-n1)).
sets n2: (n-n1).
xapply drop_spec as l2.
xapplys. fold n2.
xapply~ as l2' [P2 S2].
  subst n2 l2. rewrite~ drop_length.
xapply~ as l1' [P1 S1].
xapply~ (merge_spec _A Le) as l'.
xreturns. rewrite~ (@cutting_int _ n n1 n2 l).
  applys sorts_permut of_hyps Pl'.
  apply~ permut_app_lr. rewrite~ <- Pl2.
(* -- main call -- *)
xapply length_spec as len. xred.
xapplys. xcase.
  xreturns. destruct~ l. destruct~ l.
   calc_length in Plen. subst len. false.
  xapply as l'. subst len.
   xreturns. rewrite abs_pos_nat in Pl'.
    rewrite~ (@truncate_length _ l) in Pl'.
Qed.
```

Figure 5. Specification and verification of the merge sort function

specification is polymorphic in the type A, which is the type of the logical representation of items in the list, and is also polymorphic in _A, which is intended to be the associated encoder for values of type A. The specification is also parametrized by a binary relation, written le, and a proof that this relation is a total pre-order relationship on values of type A.

The core of the specification follows. It states that the application of merge_sort to a function cmp and to a list l is a term, written t, whose behaviour is described as follows: if cmp is a *comparison function* that implements the order le, then t returns a list which is the result of sorting l with respect to le. A comparison function implements the order relation le under the following condition: for any two arguments, it should return an integer less or equal to zero if and only if its first argument is smaller than its second argument, relatively to le. The formal definition that caracterizes comparsion functions is stated as follows:

```
Definition comparator
 (cmp:Val) (A:Type) (_A:A->Val) (le:relation A) :=
    spec cmp [x1:_A] [x2:_A] is
      |> Int | (fun n =>  n <= 0 <-> le x1 x2).
```

The last line of the specification states that the value returned on a call to merge_sort is a list l' such that the proposition (sorts A le l l') holds. Note that the name l' does not appear in the specification due to the use of a partial application of the predicate sorts, and that the type A is left out since it can be inferred by Coq's implicit arguments mechanism. The predicate sorts is defined in terms of list permutation and list sortedness, as follows:

```
Definition sorts
 (A:Type) (le:relation A) (l l':list A) :=
    permut l l' /\ sorted le l'.
```

The verification proof script follows the specification lemma. The structure of the proof closely matches the structure of the source program. As for any Coq proof, a verification script consists in a sequence of tactic invocations, separated with dots. The tactics that appear in a verification script fall in two categories. On the one hand, the script contains tactics devoted to applying the reasoning rules that we have introduced earlier on (in §4). These tactics are provided by our library, and we call them *administrative tactics* in the rest of the paper. These administrative tactics can be easily recognize as their names always start with the character 'x'. On the other hand, the script contains calls to standard general-purpose Coq tactics.[4] These tactics are used to argue, at the logical level, for the correctness of the program. When general-purpose tactics are used, the details of the deep embedding, and in particular the encoding of values, are no longer involved. Although it is not an easy task to describe on paper the working of an interactive proof, we try and describe the main ingredients that appear in the verification script of the merge sort function, starting with a brief description of the administrative tactics. Remark: when a tilde symbol (~) follows the name of a tactic, it simply indicates a call to auto, the automated proof-search feature of Coq, on all subgoals.

The tactic xintros introduces the arguments of a function. It corresponds to the rule SPEC-INTRO. xred is used to contract a let-expression. It implements the rule REDUCTION. Similarly, xcase, xpat and xredfail are used to reduce pattern matchings. xreds iterates xred as many time as possible. xinfun is used to specify a local function (here, the recursive function sort), and the verification of this local function follows as first subgoal. The tactic xinduction applies the rule SPEC-INDUCTION. In Figure 5, the arguments of the call to xinduction indicate that the value of the first parameter of the function has an integer value that decreases on each recursive calls. More generally, any well-founded relation can be used to argue for termination. The tactic xreturns is used to prove the behaviour of a term reduced to a value. The tactic xapply is used to reason on an application. It combines the rules SPEC-ELIM and CTX-RETURNS. Notice that xapply is often followed with a list of identifiers. These identifiers are used to give explicit names to intermediate results. The tactic xapplys is similar, except that it substitutes the result of the application directly into the current term, rather than naming it.

Our proof script contains a few lines devoted to the setting up of the case analysis for the pattern matching that occurs in the source code of the function sort. These five lines are located just before the case analysis on n, starting with the keyword "asserts". The intermediate lemma being proved there simply describes the various possible cases. When the pattern matching is simple enough, one

[4] The Coq expert will notice that we rely on a number of extensions to the set of builtin Coq tactics. Our extended set of general-purpose tactics, which aims for improved tactic behaviour, is independent from this work.

needs not explicitly state such a lemma, since the builtin case analysis tactic of Coq works just fine. However, more complex patterns do require this kind of lemma, which we state and prove by hand for the time being, but are looking forward to generate in the future.

Administrative tactics structure the proof in a way that reflects the structure of the program. They allow to explicitly assign names to variables and hypotheses that need to be mentioned in the formal reasoning. Naming properly variables and hypotheses on the one hand, and keeping track of the structure of proofs on the other hand, are two ingredients critical to script maintainability. In particular, our proof scripts are not affected by alpha-conversion in the source code, and a local modification in a source program can be accomodated with a local modification at the corresponding position in its verification script.

Outside administrative tactics, there is some reasoning involved for establishing that the pre-condition of functions called are satisfied, using the facts gathered from prior function calls and case analyses. Most of this work is handled through automation, but sometimes explicit rewriting or invocation of external lemmas is necessary. For example, to establish the correctness of the merge sort function, one needs to justify that the concatenation of the n first elements of a list l with the sublist obtained by cutting out the n first elements of the same list l is exactly l.

5.3 Caml's list library

The first interest in considering a list library lies in the fact that it is a very good sample of idiomatic functional code. In particular, its code relies heavily on pattern matching and recursion. Also, polymorphic higher-order functions are used extensively to implement iterators. Furthermore, a typical functional program is very likely to make use of lists and list operations defined in the library, so the verification of functions from the list library is a requirement for the complete verification of a program that uses these functions. In what follows, we describe the main patterns that arose in the specification of the 34 functions included in OCaml's list library. Statistics on the size of specifications and proofs appear in the next section.

Functions such as length (which computes the length of a list), append (which concatenates two lists) or concat (which flattens a list of lists) are specified using their logical counterpart directly. Consider for example the specification of the function length:

```
Lemma length_spec : forall (A:Type) (_A:A->Val),
  spec length [l:_A] is (|> _Int | = Coq.length l)
```

This specification states that when function length is applied to the encoding of a logical list l, it returns the encoding of an integer equal to the logical length (Coq.length) of the length of list l.

Functions such as combine (which builds a list of pairs from two lists of equal length) or nth (which returns the n-th element of a list) are also specified using their logical counterpart, but only under the hypothesis that the arguments satisfy a given pre-condition. For instance, nth implements Coq.nth provided it is applied to a non-negative index strictly smaller than the length of the list.

```
Lemma nth_spec : forall A _A,
  spec nth [l:_List _A] [n:_Int] = t is
    (0 <= n < Coq.length l) ->
    t |> _A | = (Coq.nth n l)
```

A higher-order function such as map (which applies a function to all the values in a list and returns the list of the results) can be given a simple specification when the function f that it takes as argument is naturally reflected as a function F from the logic, that is, when f implements F. This specification of map states that the application of this function to f and to the encoding of a list l produces a list equal to the encoding of the result of the application of Coq.map to F and l.

```
Lemma map_spec : forall A _A B _B,
  spec map [f:_Val] [l:_List _A] = t is
    forall (F:A->B),
    (spec f [x:_A] is |> _B | = F x) ->
    t |> _List _B | = (Coq.map F l).
```

Yet, this specification is not the most general one. Indeed, we should not suppose that the function f implements some logical function F, but only assume that there exists some post-condition that relates the inputs and the outputs of f. Moreover, we should only require information on the behaviour of f on the set of values that actually occur in the list l. More precisely, we suppose that there exists a post-condition Q such that for any item x (of type A) in the list l, function f terminates on the input x and returns a result y (of type B) such that $(Q\,x\,y)$:

```
  spec f [x:_A] = t is
    (Coq.mem x l) -> (t |> _B | (fun y => Q x y))
```

If the above is true, then the application of function map to the function f and to the encoding of the list l returns a list, call it l', such that pairs of elements at corresponding indices in l and l' satisfy predicate Q. This inductively-defined property is captured by the proposition "Coq.for_all2 $Q\,l\,l'$". The more general specification of map appears below.

```
Lemma map_spec' : forall A _A B _B,
  spec map [f:_Val] [l:_List _A] = t is
    forall (Q:A->B->Prop),
    (spec f [x:_A] = t' is
      (Coq.mem x l) -> (t' |> _B | Q x)) ->
    t |> _List _B | (Coq.for_all2 Q l)
```

Functions such as fold_right are specified using invariants. Given a function f, a value a and a list $\langle x_1; x_2; \ldots; x_n\rangle$, fold_right computes "$f\,x_1\,(f\,x_2\,\ldots(f\,x_n\,a)\ldots)$". If a binary predicate I holds of the empty list and of the initial value a of the accumulator, and if it is preserved at each step in the folding process, then the invariant I holds of the entire list given to fold_right and of the final result. Technically:

```
Lemma fold_right_spec : forall A _A B _B,
  spec fold_right [f:_Val] [l:_List _A] [a:_B]
                                        = t is
  forall (I : List A -> B -> Prop),
  I nil a ->
  (spec f [x:_A] [b:_B] = t' is
    forall l', I l' b -> t' |> _B | I (x::l')) ->
  t |> _B | (I l)
```

Finally, Caml's list library includes a 65-line long implementation of merge sort. Compared to the function presented in the example earlier on (§5.2), the library function is carefully optimized for performance. By manipulating lists sorted either in ascending or descending order, it minimizes the number of cells allocated at runtime. This leads to a longer and more complex code, whose verification is naturally more challenging. The proof-checking of this optimized implementation of merge sort takes about 30 seconds (measured on a bi-processor 2.4Ghz machine), which accounts for half of the time needed to verify the entire list library.

5.4 Bytecode compiler and interpreter

The matter of this second case study is the complete verification of a virtual machine for a core ML language. This development includes three functions. First, function "compile" compiles a λ-terms into bytecode. Second, function "run" interprets a sequence

	Source code	Specification	Verification
List.length	3	1	4
List.map	3	4	4
List.fold_right	4	6	4
List.split	4	2	5
List.merge_sort	65	5	298
OCaml's list library	201	143	558
ML virtual machine	43	8	95

Figure 6. Size of case studies (non-empty lines)

of bytecode. Third and last, function "execute" combines the two previous functions to implement a relatively efficient virtual machine. The source code is made of 24 lines of type declarations plus 19 lines of actual code, and relies heavily on pattern-matching. Both the code and the proof of correctness were adapted from work by Leroy [12] which focuses on difficulties arising in the formalization of an ML compiler.

The specification of execute states that if a program p computes a value with respect to a call-by-value semantics, then the application of execute to the code of the program p returns this value. Verifying this specification is nontrivial because the argumentation of why the virtual machine appropriately simulates its input is quite remote from what appears in the source code of this machine. In particular, the termination of the bytecode interpreter has to be established by induction on the finite reduction sequence of the source program, although the code of the source program is thrown away by the virtual machine after the compilation phase.

We have completely separated the logical argumentation of why the virtual machine is correct, which uses logical definitions of the machine states and transitions, from the actual verification of the source code, for which we only need to verify that the code correctly implements the set of transitions. The former part requires 71 lines of definitions and proofs, while the latter requires 8 lines of specification and 24 lines of proofs.

6. Implementation

We have implemented a tool that parses source code in Caml syntax and generates Coq definitions for type declarations, for data constructors, for encoders, and for each top-level value defined in the source. This program is made of about 800 lines of OCaml, not counting a copy of OCaml's parser.

The Coq library for the deep embedding which we have set up contains: (1) definitions of the syntax of the embedded language, (2) pretty-printing directives for displaying embedded syntax during interactive proofs, (3) definition of the semantics, of behaviours and of specification predicates, (4) statements and proofs of the reasoning rules, (5) definition of the tactics that help applying these rules, and (6) definition and verification of a number of basic functions (e.g. negation, disequality, etc…). There are about one thousand lines of definitions and notation, another thousand lines for proofs, and about five hundred lines of tactic definitions.

As explained earlier on (§5.2), proof steps fall in two categories: administrative steps, for navigating through the structure of programs, and high-level arguments, which explain why programs work. For administrative steps, we observe that the number of administrative tactic involved is in average equal to the number of nodes in the abstract syntax tree of this program. Considering that we are working with a high-level programming language in which programs typically have a rather concise code, it seems reasonable to expect programs to require proofs longer than their code. So, starting with an overhead equal to the size of the code should not be prohibitive.

Practical evidence of the relative brevity of our scripts is given in Figure 6. It contains statistics on the number of lines of code, of specifications and of proofs for selected functions as well as for complete developments. It appears that, for a simple function, the size of the proof of a program is typically no longer than the sum of the size of its code plus the size of its specification. Of course, more complex functions require longer proofs, but this happens independently of the framework being used.

7. Related work

Tools such as Spec# [2] for C# programs, and tools using JML [3] as specification language for Java programs such as Krakatoa [14], are serious attempts at setting up a verification process for industrial programming languages. Yet, these tools impose specifications to be expressed in first-order logic. As a consequence, they fail to scale up due to the lack of support for higher-order functions, as well as lack of modularity and abstraction. Moreover, despite recent progress, automated provers still have difficulties discharging proof obligation of nontrivial properties, and often fail to explain where additional or stronger annotations are required. Caduceus [9], based on the framework Why [8], allows for the verification of C programs. It supports annotation with higher-order logic predicates, but does not support higher-order functions. Proof obligations that are not discharged automatically can be discharged using an interactive theorem prover. However, the statement of these proof obligations is typically large and clumsy, involving dozens of machine generated names, making proofs difficult to carry out and to maintain.

Pangolin [21] is a tool that successfully applies the Hoare-style methodology to a purely-functional language, including support for polymorphic higher-order functions. In this setting, programs are written in a syntax close to Caml and annotated with higher-order logic specifications and invariants. Proof obligations are then extracted using a VCG and can be discharged either automatically or interactively using Coq. As for the aforementioned tools, Pangolin checks only partial correctness of well-typed programs. The most fundamental difference with our work lies in the way in which computational entities are lifted to the logical level. In Pangolin values are lifted automatically at the time of generating proof obligations, while we do it explicitly through the use of encoders. Furthermore functions are reflected by a pair of a pre-condition and a post-condition, and not by their code. Thus, functions, contrary to other values, are described in Pangolin's logic by a pair of propositions (in Prop) and not as a datatype (in Set). It is unclear what happens when functions are stored in data-structures and whether it is possible to verify code which exploits this.

Reflection of values is trivially dealt with by a shallow embedding: program values and logical values are simply identified. The downside of this identification is that programming functions undergo the same restrictions as logical functions. As explained in the introduction, this leads to constraints with respect to partiality, recursion and exceptions. In short, the shallow embedding approach benefits from a very low cost of entry, but it requires significant effort for making a given source code fit the framework. Some people, among which the authors of Epigram [15], argue that, after all, it might be a good thing to rethink the way we write our programs. Indeed, writing the code in a style that explicitly establishes data structure invariants would certainly ease program verification (and, in particular, simplify the task of type-checking in presence of dependent types). Yet, it is unclear how many programmers would be willing to make such a dramatic change to their programming style.

Ynot [19] can be seen as an attempt to to overcome these limitations on the source language when using a shallow embedding. It axiomatically extends Coq with a monad that encapsulates all impurities: non-termination, exceptions, and side-effects. Thus, programs are written in terms of Coq language constructs plus the extra monadic operators (it is still a shallow embedding), and executable

Haskell code can be extracted. The reasoning is carried out with an axiomatized predicate that captures Hoare-logic specifications and integrates a notion of separation for local reasoning on side-effects. Impressive examples, such as a set of imperative implementations of finite maps, demonstrate the theoretical strength of the approach. One visible difference with our approach is that Ynot requires source programs to be well-typed. Also, because Ynot carves in stone a monadic type for Hoare triples, there is no lightweight solution for extending it to support a richer programming language nor to enable verification of total correctness properties.

Mehta and Nipkow have investigated the use of a deep embedding for reasoning on programs written in a simple imperative language, with assignment, sequence, conditional and while loops. Hoare-style reasoning rules are proved correct with respect to the semantics of the language, and a tactic implements a simple VCG to produce proof obligations. For the VCG tactic to work, the source code needs to be annotated with all its invariants. In our work, invariants only appear in proof scripts. In fact, many of them need not be written explicitly since they can be deduced from the specification of the functions being applied in the code. Mehta and Nipkow start by using a predicate to relate list- or tree-shaped data structures that lie in the store with their logical counterpart. Later in their paper, they implement this predicate using a function that corresponds to what we have called a decoder. They illustrate their approach with a formalization of the Schorr-Waite graph marking algorithm, a relatively short program (a dozen of lines of code) with nontrivial invariants.

XCAP [20] and SCAP [7] are frameworks for reasoning about assembly programs. The most notable similarity with our work is the use of a deep embedding of the target language. These two frameworks have been used to verify short but fairly complex assembly routines. The verification involves hundreds of lines of proofs per machine instruction. The major difference with our work is that XCAP and SCAP target a low-level imperative language, whose mechanisms are quite far from those of a high-level functional language. As argued by their authors [6], a central advantage of relying on a deep embedding is that it allows for interoperability between different program logics (e.g. XCAP and SCAP) by ultimately defining all the logics used in terms of machine code semantics. Interoperability is particularly useful when setting up a certified operating system, since the components involved rely on different computation features and span different abstraction levels.

8. Conclusion

We have set up a technology for specifying and proving total correctness of purely functional programs. It relies on three main ingredients: a deep embedding of the programming language, a family of encoders to implement the reflection between program values and logical values, and tactics to implement the application of big-step reasoning rules. Our approach is relatively lightweight, very expressive, and can be used to produce robust proofs of complex pieces of functional code.

Acknowledgments

I wish to thank François Pottier for fruitful discussions on program verification and careful reading of this paper, Xavier Leroy for technical discussions related to non-termination and co-induction, Didier Rémy for helping to improve the presentation, and Jean-Baptiste Tristan for valuable feedback on early drafts of this paper.

References

[1] Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *POPL*, January 2008.

[2] Mike Barnett, Rob DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *JOT*, 3(6), 2004.

[3] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *STTT*, 7(3):212–232, June 2005.

[4] Arthur Charguéraud and François Pottier. Functional translation of a calculus of capabilities. In *ICFP*, September 2008.

[5] The Coq Development Team. The Coq proof assistant reference manual, version 8.1. At `http://coq.inria.fr/`, 2007.

[6] Xinyu Feng, Zhaozhong Ni, Zhong Shao, and Yu Guo. An open framework for foundational proof-carrying code. In François Pottier and George C. Necula, editors, *TLDI*, pages 67–78. ACM, 2007.

[7] Xinyu Feng, Zhong Shao, Alexander Vaynberg, Sen Xiang, and Zhaozhong Ni. Modular verification of assembly code with stack-based control abstractions. In Michael I. Schwartzbach and Thomas Ball, editors, *PLDI*, pages 401–414. ACM, 2006.

[8] Jean-Christophe Filliâtre. Verification of non-functional programs using interpretations in type theory. *JFP*, 13(4):709–745, 2003.

[9] Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of C programs. In *Formal Methods and Software Engineering, 6th ICFEM 2004*, volume 3308 of *LNCS*, pages 15–29. Springer-Verlag, 2004.

[10] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[11] Peter V. Homeier and David F. Martin. Trustworthy tools for trustworthy programs: A verified verification condition generator. In Thomas F. Melham and Juanito Camilleri, editors, *TPHOLs*, volume 859 of *LNCS*, pages 269–284. Springer, 1994.

[12] Xavier Leroy. Coinductive big-step operational semantics. In Peter Sestoft, editor, *ESOP*, volume 3924 of *Lecture Notes in Computer Science*, pages 54–68. Springer Verlag, 2006.

[13] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL*, pages 42–54, January 2006.

[14] Claude Marché, Christine Paulin Mohring, and Xavier Urbain. The Krakatoa tool for certification of Java/JavaCard programs annotated in JML. *JLAP*, 58(1–2):89–106, 2004.

[15] Conor McBride and James McKinna. The view from the left. *JFP*, 14(1):69–111, 2004.

[16] Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. In Franz Baader, editor, *CADE*, volume 2741 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 2003.

[17] Michael J.C. Gordon. Mechanizing programming logics in higher-order logic. In G.M. Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automatic Theorem Proving*, pages 387–439. Springer-Verlag, 1988.

[18] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and separation in hoare type theory. In *ICFP*, pages 62–73, 2006.

[19] Aleksandar Nanevski, Greg Morrisett, Avi Shinnar, Paul Govereau, and Lars Birkedal. Ynot : Reasoning with the awkward squad. In *ICFP*, September 2008.

[20] Zhaozhong Ni and Zhong Shao. Certified assembly programming with embedded code pointers. In *POPL*, pages 320–333, 2006.

[21] Yann Régis-Gianas and François Pottier. A Hoare logic for call-by-value functional programs. In *MPC*, July 2008.

[22] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.