Program Verification Through Characteristic Formulae

Arthur Charguéraud
INRIA
arthur.chargueraud@inria.fr

Abstract

This paper describes CFML, the first program verification tool based on characteristic formulae. Given the source code of a pure Caml program, this tool generates a logical formula that implies any valid post-condition for that program. One can then prove that the program satisfies a given specification by reasoning interactively about the characteristic formula using a proof assistant such as Coq. Our characteristic formulae improve over Honda *et al*'s *total characteristic assertion pairs* in that they are expressible in standard higher-order logic, allowing to exploit them in practice to verify programs using existing proof assistants. Our technique has been applied to formally verify more than half of the content of Okasaki's *Purely Functional Data Structures* reference book.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Formal methods

General Terms Verification

1. Overview

1.1 Introduction to characteristic formulae

This paper describes an effective technique to formally specify and verify the source code of an existing purely functional program. The key idea is to generate, in a systematic manner, a logical formula for each top-level definition from the source program. Those formulae, expressed solely with standard higher-order logic connectives, carry a precise account of what the program does. Verification of the program can then be conducted by reasoning on its characteristic formula using an off-the-shelf proof assistant.

For the sake of example, consider the following recursive function, which divides by two any non-negative even integer.

$$\begin{array}{l}\mathsf{let\ rec\ half}\ x = \\ \quad \mathsf{if}\ x = 0\ \mathsf{then}\ 0 \\ \quad \mathsf{else\ if}\ x = 1\ \mathsf{then\ fail} \\ \quad \mathsf{else\ let}\ y = \mathsf{half}\,(x-2)\ \mathsf{in} \\ \quad\quad y+1\end{array}$$

The corresponding characteristic formula appears next. Given an argument x and a post-condition P, the characteristic formula for half describes what needs to be proved in order to establish that the application of half to x terminates and returns a value satisfying the predicate P, written "$\mathsf{AppReturns\,half}\,x\,P$".

$$\begin{array}{l}\forall x.\,\forall P. \\ \left(\begin{array}{l} \quad (x = 0 \Rightarrow P\,0) \\ \wedge\ (x \neq 0 \Rightarrow \\ \quad\quad (x = 1 \Rightarrow \mathsf{False}) \\ \quad \wedge\ (x \neq 1 \Rightarrow \\ \quad\quad\quad \exists P'.\quad (\mathsf{AppReturns\,half}\,(x-2)\,P') \\ \quad\quad\quad\quad\quad \wedge\ (\forall y.\,(P'\,y) \Rightarrow P\,(y+1)) \quad)) \end{array}\right) \\ \Rightarrow\ \mathsf{AppReturns\,half}\,x\,P\end{array}$$

When x is equal to zero, the function half returns zero. So, if we want to show that half returns a value satisfying P, we have to prove "$P\,0$". When x is equal to one, the function half crashes, so we cannot prove that it returns any value. The only way to proceed is to show that the instruction fail cannot be reached. Hence the proof obligation False. Otherwise, we want to prove that "$\mathsf{let}\ y = \mathsf{half}\,(x-2)\ \mathsf{in}\ y+1$" returns a value satisfying P. To that end, we need to exhibit a post-condition P' such that the recursive call to half on the argument $x-2$ returns a value satisfying P'. Then, for any name y that stands for the result of this recursive call, assuming that y satisfies P', we have to show that the output value $y+1$ satisfies the post-condition P.

More generally, the characteristic formula $[\![t]\!]$ associated with a term t can be used to prove that this term returns a value satisfying a particular post-condition. For any post-condition P, the term t terminates and returns a value satisfying P if and only if the proposition "$[\![t]\!]\,P$" is true. The application "$[\![t]\!]\,P$" is a standard higher-order logic proposition that can be proved using an off-the-shelf proof assistant. Thus, characteristic formulae can be used in practice to verify that a program satisfies its specification.

For program verification to be realistic, the proof obligation "$[\![t]\!]\,P$" should be easy to read and manipulate. Fortunately, our characteristic formulae can be pretty-printed in a way that closely resemble source code. For example, the characteristic formula associated with half is displayed as follows.

$$\begin{array}{l}\mathsf{LET\ half}\ :=\ \mathsf{Fun}\ x \mapsto \\ \quad \mathsf{If}\ x = 0\ \mathsf{Then\ Return}\ 0 \\ \quad \mathsf{Else\ If}\ x = 1\ \mathsf{Then\ Fail} \\ \quad \mathsf{Else\ Let}\ y\ :=\ \mathsf{App\ half}\,(x-2)\ \mathsf{In} \\ \quad\quad \mathsf{Return}\ (y+1)\end{array}$$

At first sight, it might appear that the characteristic formula is merely a rephrasing of the source code in some other syntax. To some extend, this is true. A characteristic formula is a sound and complete description of the behaviour of a program. Thus, it carries no more and no less information than the source code of the program itself. However, characteristic formulae enable us to move away from program syntax and conduct program verification entirely at the logical level. Characteristic formulae thereby avoid all the technical difficulties associated with manipulation of program syntax and make it possible to work directly in terms of higher-order logic values and formulae.

ICFP'10, September 27–29, 2010, Baltimore, Maryland, USA.

1.2 Specification and verification

One of the key ingredient involved in characteristic formulae is the predicate AppReturns, which is used to specify functions. Because of the mismatch between program functions, which may fail or diverge, and logical functions, which must always be total, we cannot represent program function using logical functions. For this reason, we introduce an abstract type, named Func, which we use to represent program functions. Values of type Func are exclusively specified in terms of the predicate AppReturns. The proposition "AppReturns $f\ x\ P$" states that the application of the function f to an argument x terminates and returns a value satisfying P. Hence the type of AppReturns, shown below.

$$\mathsf{AppReturns} : \forall A\ B.\ \mathsf{Func} \to A \to (B \to \mathsf{Prop}) \to \mathsf{Prop}$$

Remark: an OCaml function f of type $A \to B$ is described in Coq at the type Func, regardless of what A and B might be. This is not a problem because propositions of the form "AppReturns $f\ x\ P$" can only be derived when x has type A and P has type $B \to \mathsf{Prop}$.

The predicate AppReturns is used not only in the definition of characteristic formulae but also in the statement of specifications. One possible specification for half is the following: if x is the double of some non-negative integer n, then the application of half to x returns an integer equal to n. The corresponding higher-order logic statement appears next.

$$\forall x.\ \forall n.\ n \geq 0 \ \Rightarrow\ x = 2 * n \ \Rightarrow\ \mathsf{AppReturns}\ \mathsf{half}\ x\ (= n)$$

Remark: the post-condition $(= n)$ is a partial application of equality: it is short for "$\lambda a.\ (a = n)$". Here, the value n corresponds to a ghost variable: it appears in the specification of the function but not in its source code. The specification that we have considered for half might not be the simplest one, however it illustrates our treatment of ghost variables.

Our next step is to prove that the function half satisfies its specification using its characteristic formula. We first give the mathematical presentation of the proof and then show the corresponding Coq proof script. The specification is to proved by induction on x. Let x and n be such that $n \geq 0$ and $x = 2{*}n$. We apply the characteristic formula to prove "AppReturns half $x\ (= n)$". If x is equal to 0, we conclude by showing that n is equal to 0. If x is equal to 1, we show that $x = 2{*}n$ is absurd. Otherwise, $x \geq 2$. We instantiate P' as "$= n - 1$", and prove "AppReturns half $(x - 2)\ P'$" using the induction hypothesis. Finally, we show that, for any y such that $y = n - 1$, the proposition $y + 1 = n$ holds. This completes the proof. Note that, through this proof by induction, we have proved that the function half terminates on its domain.

Formalizing the above piece of reasoning in a proof assistant is straightforward. In Coq, a proof script takes the form of a sequence of tactics, each tactic being used to make some progress in the proof. The verification of the function half could be done using only built-in Coq tactics. Yet, for the sake of conciseness, we rely on a few specialized tactics to factor out repeated proof patterns. For example, each time we reason on a "if" statement, we want to split the conjunction at the head of the goal and introduce one hypothesis in each subgoal. The tactics specific to our framework can be easily recognized: they start with the letter "x". The verification proof script for half appears next.

```
xinduction (downto 0).
xcf. introv IH Pos Eq. xcase.
  xret. auto.                    (* x = 0 *)
  xfail. auto.                   (* x = 1 *)
  xlet.                          (* otherwise *)
    xapp (n-1); auto.            (* half (x-2) *)
    xret. auto.                  (* return y+1 *)
```

The interesting steps in that proof are: the setting up of the induction on the set of non-negative integers (xinduction), the application of the characteristic formula (xcf), the case analysis on the value of x (xcase), and the instantiation of the ghost variable n with the value "$n - 1$" when reasoning on the recursive call to half (xapp). The tactic auto runs a goal-directed proof search and may also rely on a decision procedure for linear arithmetic. The tactic introv is used to assign names to hypotheses. Such explicit naming is not mandatory, but in general it greatly improves readability of proof obligations and robustness of proof scripts.

When working with characteristic formulae, proof obligations always remain very tidy. The Coq goal obtained when reaching the subterm "$\mathsf{let}\, y = \mathsf{half}\,(x - 2)\,\mathsf{in}\, y + 1$" is shown below. In the conclusion (stated below the line), the characteristic formula associated with that subterm is applied to the post-condition to be established $(= n)$. The context contains the two pre-conditions $n \geq 0$ and $x = 2 * n$, the negation of the conditionals that have been tested, $x \neq 0$ and $x \neq 1$, as well as the induction hypothesis, which asserts that the specification that we are trying to prove for half already holds for any non-negative argument x' smaller than x.

```
x : int
IH : forall x', 0 <= x' -> x' < x ->
     forall n, n >= 0 -> x' = 2 * n ->
     AppReturns half x' (= n)
n : int
Pos : n >= 0
Eq : x = 2 * n
C1 : x <> 0
C2 : x <> 1
------------------------------------------------
(Let y := App half (x-2) in Return (1+y)) (= n)
```

As illustrated through the example, a verification proof script typically interleaves applications of "x"-tactics with pieces of general Coq reasoning. In order to obtain shorter proof scripts, we set up an additional tactic that automates the invokation of x-tactics. This tactic, named xgo, simply looks at the head of the characteristic formula and applies the appropriate x-tactic. A single call to xgo may analyse an entire characteristic formula and leave a set of proof obligations, in a similar fashion as a Verification Condition Generator (VCG).

Of course, there are pieces of information that xgo cannot infer. Typically, the specification of local functions must be provided explicitly. Also, the instantiation of ghost variables cannot always be inferred. In our example, Coq automation is slightly too weak to infer that the ghost variable n should be instantiated as $n - 1$ in the recursive call to half. In practice, xgo will stop running whenever it lacks too much information to go on. The user may also explicitly tell xgo to stop at a given point in the code. Moreover, xgo accepts hints to be exploited when some information cannot be inferred. For example, we can run xgo with the indication that the function application whose result is named y should use the value $n - 1$ to instantiate a ghost variable. In this case, the verification proof script for the function half is reduced to:

```
xinduction (downto 0). xcf. intros.
xgo~ 'y (Xargs (n-1)).
```

Note that automation, denoted by the tilde symbol, is able to handle all the subgoals produced by xgo.

For simple functions like half, a single call to xgo is usually sufficient. However, for more complex programs, the ability of xgo to be run only on given portions of code is crucial. In particular, it allows one to stop just before a branching point in the code in order to establish facts that are needed in several branches. Indeed, when a piece of reasoning needs to be carried out manually, it is

extremely important to avoid duplicating the corresponding proof script across several branches.

To summarize, our approach allows for very concise proof scripts whenever verifying simple pieces of code, thanks to the automated processing done by `xgo` and to the good amount of automation available through the proof search mechanism and the decision procedures that can be called from Coq. In the same time, when verifying more complex code, our approach offers a very fine-grained control on the structure of the proofs and it greatly benefits from the integration in a proof assistant for proving non-trivial facts interactively.

1.3 Implementation

Our implementation is named CFML, an acronym for *Characteristic Formulae for ML*. It parses an OCaml source code and normalizes its syntax, making sure that applications and function definitions be bound to a name. Our tool then type-checks the code and produces a set of Coq definitions. For each type definition in the source program, it generates the corresponding definition in the logic. For each top-level value definition, it introduces one abstract variable to represent the result of the evaluation of this definition, plus one axiom stating the characteristic formula associated with the definition. For example, for the program "$\mathsf{let}\, x = \mathsf{let}\, y = 2\, \mathsf{in}\, y * y$", we generate a first axiom, named x, of type int, and a second axiom with a type of the form "$\forall P. [\ldots] \Rightarrow P\, x$". This characteristic formula for x describes what needs to be proved in order to establish that x satisfies a given predicate P.

We have proved on paper that characteristic formulae are sound with respect to the logic of Coq, by showing that those axioms could be realized in Coq, at least in theory. (In practice, generating actual proof terms would require a lot of effort, so we have not implemented it.) Moreover, in order to preserve soundness, each time we introduce an axiom to represent a value we generate a proof that the type of this value is inhabited. For example, our tool rejects the program definition "$\mathsf{let}\, x = \mathsf{fail}$" because the type $\forall A.A$ cannot be proved to be inhabited. Rejecting this kind of program is not really a limitation since it would not be possible anyway to prove that such a program returns a value.

For the time being, only purely functional programs are supported. However, we strongly believe that characteristic formulae can be extended with heap descriptions and frame rules, without compromising the possibility of pretty-printing characteristic formulae like source code. We leave the extension to side-effects to future work and focus in this paper on demonstrating the benefits of characteristic formulae for reasoning on pure programs.

This paper is organized as follows. First, we explain how our approach compares against existing program verification techniques (§2). Second, we describe formalizations of purely functional data structures (§3). Third, we describe the algorithm for generating characteristic formulae (§4), and formally define our specification predicates (§5). Finally, we discuss the soundness and completeness of characteristic formulae (§6), and conclude (§7).

2. Comparison with related work

2.1 Characteristic formulae

The notion of *characteristic formula* originates in process calculi. Given the syntactic definition of a process, the idea is to generate a temporal logic formula that precisely describes that process [12, 17, 23]. In particular, behavioural equivalence or dis-equivalence of two processes can be established by comparing their characteristic formulae. Such a proof can be conducted in temporal logic rather than through reasoning on the syntactic definition of the processes.

In a similar way, the characteristic formula of a program is a logical formula that carries a precise description of this program, without referring to its syntactic definition. For the sake of reasoning on functional correctness, programs can be studied in terms of their most-general specification. The theoretical insight that any program admits a most-general Hoare triple which entails all other correct specifications is nearly as old as Hoare logic. Gorelick [9] proved that every program admits a weakest pre-condition (the minimum requirement to ensure safe termination) and a strongest post-condition (the maximal amount of information that can be gathered about the output of the program).

The suggestion that most-general specifications could be exploited to verify programs first appears, as far as we know, in recent work by Honda, Berger and Yoshida [10]. The authors consider a particular Hoare logic and exhibit an algorithm for constructing the *total characteristic assertion pair* (TCAP) of a program, which corresponds to most-general Hoare triple. TCAPs offer an alternative way of proving that a program satisfies a given specification: rather than building a derivation using the reasoning rules of the Hoare program logic, one may simply prove that the pre-condition of the specification implies the weakest pre-condition and that the post-condition of the specification is implied by the strongest post-condition. The verification of those two implications can be conducted entirely at the logical level. Our work builds upon a similar idea, relying on characteristic formulae to move away from program syntax and carry out the reasoning in the logic.

Our main contribution is to express the characteristic formula of a program in terms of a standard higher-order logic. By contrast, TCAPs are expressed in an ad-hoc logic. In particular, the values from this logic are well-typed PCF values, including first-class functions. It is not immediate to translate this logic into a standard logic, because of this mismatch between program functions, which may fail or diverge, and logical functions, which must always be total. Due to the nonstandard logic it relies upon, Honda *et al*'s TCAPs cannot be manipulated in an existing theorem prover. In this work, we show how an abstract type Func can be introduced to support the ability to refer to first-class functions from the logic.

Our characteristic formulae also improve over TCAPs in that they are human-readable. While Honda *et al*'s TCAP did not fit on a screen for a program of more than a few lines, we show characteristic formulae can be displayed just like source code. The ability to read charactersitic formulae is very important in interactive proofs since the characteristic formula shows up as part of the proof obligation that the user must discharge.

2.2 Verification Condition Generators

Tools such as Spec# [1] for C# programs, Krakatoa [14] for Java programs, Caduceus [7] for C programs, Pangolin [24] for pure ML programs, and Who [11] for imperative ML programs, are all based on VCGs. They generate a set of proof obligations and rely on automated theorem provers to discharge these obligations. In the latter three systems, proof obligations that are not verified automatically can be discharged using an interactive proof assistant. However, in practice, those proof obligations are often large and clumsy, and their proofs are generally quite brittle because proof obligations are very sensitive to changes in either the source code or its invariants. In our approach, proof obligations remain tidy and can be easily related to the point of the program they arise from. Moreover, the user has the possibility to invest a little extra effort in naming hypotheses explicitly in order to be able to build very robust proof scripts.

The tool Jahob [26], which supports the verification of linked data structures implemented in a subset of Java, tries to avoid as much as possible the need for interactive proofs by annotating programs not only with their invariants but also with proof hints to guide automated theorem provers. As acknowledged by the authors, finding the appropriate hints can be very time-consuming. In

particular, one needs to compute and read the new proof obligations after any modification of a hint. Moreover, guessing hints requires a deep understanding of the VCG process and of the automated theorem provers being used. Nevertheless, there are some particular situations where providing such hints is actually very effective. Our approach naturally supports this proof technique, simply by giving the appropriate hints as argument to our tactic `xgo`. We may also set up Coq automation to apply a user-defined sequence of tactics to any proof obligation satisfying a particular pattern.

Among the tools cited above, few of them support higher-order functions: Pangolin [24] and Who [11], which combines ideas from Caduceus [7] and Pangolin [24] to handle effectfull higher-order programs. One notable difference with our work lies in the way in which functions are lifted to the logical level. In Pangolin and Who, a function is reflected in the logic as a pair of a pre-condition and of a post-condition. Instead, we reflect a function in the logic as a value of the abstract type Func and use AppReturns to specify the behaviour of this value. We believe that our approach is more appropriate when functions are given several specifications, when functions are stored in data-structures, and when higher-order functions are applied to functions specified with ghost variables.

2.3 Shallow embedding techniques

A radically different approach consists in programming directly within a theorem prover and verifying properties of the code interactively inside the same framework. Indeed, the logic of a proof assistant such as Coq is so rich that it contains a purely functional programming language. An extraction mechanism can then be used to isolate the actual source code from proof-specific elements. The shallow embedding approach can be applied in two very different styles, depending on how much types are used to enforce invariants.

The first possibility is to write programs using only basic ML types. This style is employed for instance in Leroy's formally-verified C compiler [13]. While it can be quite effective for some applications, this approach also suffers from a number of severe restrictions that restrict its scope of use. In particular, all functions must be total and recursive functions must satisfy a syntactical termination criteria. On the contrary, characteristic formulae can accommodate various syntaxes for the source language, allowing for the verification of existing programs. In particular, any (well-typed) function definition can be handled: termination does not need to be established at definition time but can be proved by induction while reasoning on the characteristic formula (the induction may be on a measure, on a well-founded relation or on any Coq predicate).

The second possibility is to write programs with more elaborated types, relying on dependent types to carry invariants (e.g. using the type "list n" to describe lists of length n). Programming with dependent types has been investigated in particular in Epigram [15], Adga [5] and Russell [25]. The latter is an extension to Coq, which behaves as a permissive source language which elaborates into Coq terms. In Russel, establishing invariants, justifying termination of recursion and proving the inaccessibility of certain branches from a pattern matching can be done through interactive Coq proofs. While Russel certainly manages to smoothen the writing of dependently-typed terms, the manipulation of dependent types remains fairly technical for non-experts. Moreover, the treatment of ghost variables remains problematic in the current implementation of Coq because extraction is not sufficiently fine-grained to erase all ghost variables. As a consequence, some ghost variables may remain in the extracted code, leading to runtime inefficiencies and possibly to incorrect asymptotic complexity.

Because they rely directly on Coq terms, the two shallow embedding approaches describe above cannot support impure programming features such as side-effects and non-termination. HTT [19], its implementation in Ynot [4] and HTT's new implementation [20] try to overcome this limitation by extending Coq with a monad in order to support effects. Like in Russel, specification appears in types. They typically take the form "STsep $P\,Q$" where P and Q describe the pre- and the post-condition in terms of heap descriptions. Verification proofs are constructed by application of Coq lemmas that correspond to the reasoning rules of the program logic. This process is partially automated through a tactic (which is implemented by reflection). In our approach, most of this work is performed during the generation of characteristic formulae, by our external tool. In the end, although the implementation strategies differ, similar kinds of proof obligations are generated. Note that the trusted base of HTT is not much smaller than ours since HTT also needs to rely on some external tool in order to extract OCaml or Haskell code from Coq scripts. Although we do not yet support side effects, we see one main advantage that characteristic formulae may have compared to HTT-based approaches in the long run. Characteristic formulae can be adapted to existing programming languages. On the contrary, following HTT's approach forces one to rewrite programs in terms of the language of Coq and of the constructors of HTT's monad. Some programming language features cannot be handled easily by HTT. For example, because pattern matching is deeply hard-wired in Coq, supporting handy features such as alias-patterns and when-clauses would be a real challenge for HTT.

A slightly different approach to shallow embeddings relies on the definition of a translation from a programming language into higher-order logic. Myreen *et al* [18] describe an effective technique for reasoning on machine code, which consists in decompiling machine code procedures into higher-order logic functions. This translation is possible only because the functional translation of a while loop is a tail-recursive function, and that nonterminating tail-recursive functions are safely accepted as logical definitions in HOL4. Lemmas proved interactively about the higher-order logic functions can then be automatically transformed into lemmas about the behaviour of the machine code. While this approach works for reasoning on machine code, it does not seem possible to apply it to programs featuring arbitrary recursion and higher-order functions.

2.4 Deep embedding techniques

A fourth approach to reasoning formally on programs consists in describing the syntax and the semantics of a programming language in the logic of a proof assistant using inductive definitions. In theory, the deep embedding approach can be applied to any programming language, it does not suffer from any limitation in terms expressiveness and it is compatible with the use of interactive proofs.

Mehta and Nipkow [16] have set up a proof of concept of a deep embedding, axiomatizing a small procedural language in Isabelle, proving Hoare-style reasoning rules, and verifying a short program using those reasoning rules. More recently, the frameworks XCAP [21] and SCAP [6] rely on deep embeddings for reasoning in Coq about assembly programs. They support reasoning on advanced patterns such as strong updates, embedded code pointers and higher-order calls. They have been used to verify short but complex assembly routines, whose proof involves hundreds of lines per instruction. Previoulsy, the author of the present paper has investigated the use of a deep embedding of the pure fragment of OCaml in Coq [2]. Characteristic formulae arose from that work, bringing major improvements.

In a deep embedding, reasoning rules of the program logic take the form of lemmas that are proved correct with respect to the axiomatized semantics of the source language. When verifying a program, those reasoning rules are applied almost in a systematic manner, following the syntax of the program. The idea that the application of those reasoning rules could be *anticipated* lead to characteristic formulae.

To illustrate this idea, consider the rule for reasoning on let-expressions in a deep embedding. The rule reads as follows: to show that "$\mathsf{let}\, x \;=\; t_1 \,\mathsf{in}\, t_2$" returns a value satisfying P, the subterm t_1 must be shown to return a value satisfying a post-condition P', and the term t_2 must be shown to return a value satisfying P under the assumption that x satisfies P'. The statement of this rule, shown below, relies on a predicate capturing that a term t returns a value satisfying a post-condition P, written "$t \Downarrow | P$". (For the sake of presentation, many technical details are omitted.)

$$\frac{t_1 \Downarrow | P' \qquad \forall x.\, P'\, x \Rightarrow t_2 \Downarrow | P}{(\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2) \Downarrow | P}$$

With characteristic formulae, the proposition "$[\![\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2]\!]\, P$" captures the fact that "$\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2$" returns a value satisfying P. This proposition is defined in terms of the characteristic formulae $[\![t_1]\!]$ and $[\![t_2]\!]$ associated with the two subterms t_1 and t_2. More precisely, "$[\![t_1]\!]\, P'$" asserts that t_1 returns a value satisfying P' and "$[\![t_2]\!]\, P$" asserts that t_2 returns a value satisfying P. Formally:

$$[\![\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2]\!]\, P \;=\; \exists P'.\; [\![t_1]\!]\, P' \;\wedge\; \forall x.\, P'\, x \;\Rightarrow\; [\![t_2]\!]\, P$$

Although this equation looks very similar to the reasoning rule, there is one important difference. With the program logic reasoning rule, the intermediate specification P' needs to be provided at the time of applying the rule. On the contrary, characteristic formulae are able to anticipate the application of the reasoning rule even without any knowledge of this intermediate specification, thanks to the existential quantification on P'. While it may appear to be fairly natural, this form of existential quantification of an intermediate specification, which takes full advantage of the strength of higher-order logic, does not seem to have been exploited in previous work.

From our experience on working on the verification of pure OCaml programs both with a deep embedding and with characteristic formulae, we conclude that moving to characteristic formulae brings at least three major improvements. First, characteristic formulae do not need to represent and manipulate program syntax. Thus, they avoid many technical difficulties, in particular those associated with the representation of binders. Also, the repeated computations of substitutions that occur during the verification of a deeply-embedded program typically lead to the generation of a proof term of quadratic size, which can be problematic for scaling up to larger programs. Second, with characteristic formulae there is no need to apply reasoning rules of the program logic manually. Indeed, the applications of those rules have been anticipated in the characteristic formulae. A practical consequence is that proof scripts are lighter and easier to automate. Third and last, characteristic formulae avoid the need to relate the deep embedding of program values with the corresponding logical values, saving a lot of technical burden. For example, consider a list of integers in a OCaml program. In the deep embedding, the description of this list is encoded using constructors from the grammar of OCaml values. With characteristic formulae, program values are translated into logical values once and for all upon generation of the formula. Thus, the list of integers would appear in the characteristic formula directly as a list of integers, significantly simplifying proofs.

The fact that characteristic formulae outperform deep embeddings is after all not a surprize: characteristic formulae can be seen as an abstract layer built on the top of a deep embedding, so as to hide uninteresting details and retain only the essence of the reasoning rules supported by the deep embedding.

3. Formalizing purely functional data structures

Chris Okasaki's book *Purely Functional Data Structures* [22] contains a collection of efficient data structures, with concise implementation and nontrivial invariants. Its code appeared as a excellent benchmark for testing the usability of our approach to program verification. So far, we have verified more than half of the contents of the book. This paper focuses on the formalization of red-black trees and give statistics on the other formalizations completed.

```
module type Fset = sig
type elem
type fset
val empty: fset
val insert: elem -> fset -> fset
val member: elem -> fset -> bool
end
```

```
module type Ordered =
sig
type t
val lt: t -> t -> bool
end
```

Figure 1. Module signatures for finite sets and ordered types

```
module RedBlackSet (Elem : Ordered) : Fset = struct
type elem = Elem.t
type color = Red | Black
type fset = Empty | Node of color * fset * elem * fset
let empty = Empty

let rec member x = function
   | Empty -> false
   | Node (_,a,y,b) ->
      if Elem.lt x y then member x a
      else if Elem.lt y x then member x b
      else true

let balance = function
  | (Black, Node (Red, Node (Red, a, x, b), y, c), z, d)
  | (Black, Node (Red, a, x, Node (Red, b, y, c)), z, d)
  | (Black, a, x, Node (Red, Node (Red, b, y, c), z, d))
  | (Black, a, x, Node (Red, b, y, Node (Red, c, z, d)))
  -> Node (Red, Node(Black,a,x,b), y, Node(Black,c,z,d))
  | (col,a,y,b) -> Node(col,a,y,b)

let rec insert x s =
  let rec ins = function
    | Empty -> Node(Red,Empty,x,Empty)
    | Node(col,a,y,b) as s ->
       if Elem.lt x y then balance(col,ins a,y,b)
       else if Elem.lt y x then balance(col,a,y,ins b)
       else s in
   match ins s with
   | Empty -> raise BrokenInvariant
   | Node(_,a,y,b) -> Node(Black,a,y,b)
end
```

Figure 2. Okasaki's implementation of Red-Black sets

Red-black trees behave like binary search trees except that each node is tagged with a color, either red or black. Those tags are used to maintain balance in the tree, ensuring a logarithmic asymptotic complexity. Okasaki's implementation appears in Figure 2. It consists of a functor that, given an ordered type, builds a module matching the signature of finite sets. Signatures appear in Figure 1.

We specify each OCaml module signature through a Coq module signature. We then verify each OCaml module implementation through a Coq module implementation that contains lemmas establishing that the OCaml code satisfies its specification. We rely on Coq's module system to ensure that the lemmas proved actually correspond to the expected specification. This strategy allows for modular verification of modular programs.

3.1 Specification of the signatures

In order to specify functions manipulating red-black trees, we need to introduce a representation predicate called rep. Intuitively, every data structure admits a mathematical model. For example, the model of a red-black tree is a set of values. Similarly, the model of a priority queue is a multiset, and the model of a queue is a se-

quence (a list). Sometimes, the mathematical model is simply the value itself. For instance, the model of an integer or of a value of type color is just the value itself.

We formalize models through instances of a typeclass named Rep. If values of a type a are modelled by values of type A, then we write "Rep a A". For example, consider red-black trees that contain items of type t. If those items are modelled by values of type T (i.e. Rep t T), then trees of type fset are modelled by values of type set T (i.e. Rep fset (set T)), where set is the type constructor for mathematical sets in Coq.

The typeclass Rep contains two constructors, as shown below. For an instance of type "Rep a A", the first field, rep, is a binary relation that relates values of type a with their model, of type A. Note that not all values admit a model. For instance, given a red-black tree e, the proposition "rep e E" can only hold if e is a well-balanced, well-formed binary search tree. The second field of Rep, named rep_unique, is a lemma asserting that every value of type a admits at most one model (we sometimes need to exploit this fact in proofs).

```
Class Rep (a:Type) (A:Type) :=
  { rep : a -> A -> Prop;
    rep_unique : forall x X Y,
       rep x X -> rep x Y -> X = Y }.
```

Remark: while representation predicates have appeared in previous work (e.g. [7, 16, 19]), our work seems to be the first to use them in a systematic manner through a typeclass definition.

Figure 3 contains the specification for an abstract finite set module named F. Elements of the sets, of type elem, are expected to be modelled by some type T and to be related to their models by an instance of type "Rep elem T". Moreover, the values implementing finite sets, of type fset, should be related to their model, of type set T, through an instance of type "Rep fset (set T)". The module signature then contains the specification of the values from the finite set module F. The first one asserts that the value empty should be a representation for the empty set. The specifications for insert and member rely on a special notation, explained next.

So far, we have relied on the predicate AppReturns to specify functions. While this works well for functions of one argument, it becomes impractical for curried functions of higher arity, in particular because we want to specify the behaviour of partial applications. So, we introduce the Spec notation, explaining its meaning informally and postponing its formal definition to §5.2. With the Spec notation, the specification of insert, shown below, reads like a prototype: insert takes two arguments, x of type elem and e of type fset. Then, for any model X of x and for any set E that models e, the function returns a finite set e' which admits a model E' equal to {X} ∪ E. (\{X} is a Coq notation for a singleton set.)

```
Parameter insert_spec :
  Spec insert (x:elem) (e:fset) |R>>
     forall X E, rep x X -> rep e E ->
     R (fun e' => exists E',
       rep e' E' /\ E' = \{X} \u E).
```

The variable R should be read as "the application of insert returns a value satisfying the following post-condition". R is bound in "|R>>"and it is applied to the post-condition of the function.

As it is often the case that arguments and/or results are described through their rep predicate, we introduce the RepSpec notation. With this new layer of syntactic sugar, the specification becomes:

```
Parameter insert_spec :
  RepSpec insert (X;elem) (E;fset) |R>>
     R (fun E' => E' = \{X} \u E  ; fset).
```

```
Module Type FsetSigSpec.
Declare Module F : MLFset. Import F.

Parameter T : Type.
Instance elem_rep : Rep elem T.
Instance fset_rep : Rep fset (set T).

Parameter empty_spec : rep empty \{}.
Parameter insert_spec :
  RepTotal insert (X;elem) (E;fset) >> = \{X} \u E ; fset.
Parameter member_spec :
  RepTotal member (X;elem) (E;fset) >> bool_of (X \in E).

End FsetSigSpec.
```

Figure 3. Specification of finite sets

```
Module Type OrderedSigSpec.
Declare Module O : MLOrdered. Import O.

Parameter T : Type.
Instance rep_t : Rep t T.
Instance le_inst : Le T.
Instance le_order : Le_total_order.

Parameter lt_spec :
  RepTotal lt (X;t) (Y;t) >> bool_of (LibOrder.lt X Y).

End OrderedSigSpec.
```

Figure 4. Specification of ordered types

The specification is now stated entirely in terms of the models, and does no longer refer to the names of OCaml input and output values. Only the type of those program values remain visible. Those type annotation are introduced by semi-columns.

The specification for the function insert given in Figure 3 makes two further simplifications. First, it relies on the notation RepTotal, which avoids the introduction of a name R when it is immediately applied. Second, we have employed for the sake of conciseness a partial application of equality, of the form "$= \{X\} \cup E$". Overall, the interest of introducing several layers of notation is that the final specifications from Figure 3 are about the simplest possible formal specifications one could hope for.

Let us describe briefly the remaining specifications. The function member takes as argument a value x and a finite set e, and returns a boolean which is true if and only if the model X of x belongs to the model E of e. Figure 4 contains the specification of an abstract ordered type module named O. Elements of the ordered type t should be modelled by a type T. Values of type T should be ordered by a total order relation. The order relation and the proof that it is total are described through instances of the typeclasses Le and Le_total_order, respectively. An instance of the strict-order relation (LibOrder.lt) is automatically derived through the typeclass mechanism. This relation is used to specify the boolean comparison function lt, defined in the module O.

3.2 Verification of the implementation

It remains to verify the implementation of red-black trees. Consider a module O describing an ordered type. Assume the module O has been verified through a Coq module named OS of signature OrderedSigSpec. Our goal is then to prove correct the module obtained by applying the functor RedBlackSet to the module O, through the construction of Coq module of signature FsetSigSpec. Thus, the verification of the OCaml functor RedBlackSet is carried through the implementation of a Coq functor named RedBlackSet-

Spec, which depends both on the module O and on its specification OS. The first few lines of this Coq functor are shown below.

```
Module RedBlackSetSpec
 (O:MLOrdered) (OS:OrderedSigSpec with Module O:=O)
 <: FsetSigSpec with Definition F.elem := O.t.
Module Import F <: MLFset := MLRedBlackSet O.
```

The next step in the construction of this functor is the definition of an instance of the representation predicate for red-black trees. To start with, assume that our goal is simply to specify a binary search tree. The rep predicate would be defined in terms of an inductive invariant called inv, as shown below. First, inv relates the empty tree to the empty set. Second, inv relates a node with root y and subtrees a and b to the set $\{Y\} \cup A \cup B$, where the uppercase variables are the model associated with their lowercase counterpart. Moreover, we need to ensure that all the elements of the left subtree A are smaller than the root Y, and that, symmetrically, elements from B are greater than Y. Those invariants are stated with help of the predicate foreach. The proposition "foreach P E" asserts that all the elements in the set E satisfy the predicate P.

```
Inductive inv : fset -> set T -> Prop :=
| inv_empty :
    inv Empty \{}
| inv_node : forall col a y b A Y B,
    inv a A -> inv b B -> rep y Y ->
    foreach (is_lt Y) A -> foreach (is_gt Y) B ->
    inv (Node col a y b) (\{Y} \u A \u B).
```

A red-black tree is a binary search tree satisfying three invariants. First, every path from the root to a leaf contains the same number of black nodes. Second, no red node can have a red child. Third, the root of the tree must be black. In order to capture the first invariant, we extend the predicate inv so that it depends on a natural number n representing the number of black nodes to be found in every path. For an empty tree, this number is zero. For a nonempty tree, this number is equal to the number m of black nodes that can be found in every path of each of the two subtrees, augmented by one if the node is black. The second invariant, asserting that a red node must have black children, can be enforced simply by testing colors. Finally, the rep predicate relates a red-black tree e with a set E if there exists a value n such that "inv n e E" holds and such that the root of e is black (the third invariant). The extended definition of inv appears in Figure 5.

In practice, we further extend the invariant with an extra boolean (this extended definition does not appear in the present paper). When the boolean is true, the definition of inv is unchanged. However, when the boolean is false, then second invariant might be broken at the root of the tree. This relaxed version of the invariant is useful to specify the behaviour of the function balance. Indeed, this function takes as input a color, an item and two subtrees, and one of those two subtrees might have its root incorrectly colored.

Figure Figure 6 shows the lemma corresponding to the verification of insert. Observe that the local recursive function ins is specified in the script. It is then verified with help of the tactic xgo.

3.3 Statistics

We have specified and verified various implementations of queues, double-ended queues, priority queues (heaps), sets, as well as sortable lists, catenable lists and random-access lists. OCaml implementations are directly adapted from Okaski's SML code [22]. All code and proofs can can be found online.[1] Figure 7 contains statistics on the number of non-empty lines in OCaml source code and in Coq scripts. The programs considered are generally short,

[1] http://arthur.charguéraud.org/research/2010/cfml/

```
Inductive inv : nat -> fset -> set T -> Prop :=
| inv_empty : forall,
    inv 0 Empty \{}
| inv_node : forall n m col a y b A Y B,
    inv m a A -> inv m b B -> rep y Y ->
    foreach (is_lt Y) A -> foreach (is_gt Y) B ->
    (n = match col with Black => m+1 | Red => m end) ->
    (match col with | Black => True
                    | Red => root_color a = Black
                         /\ root_color b = Black end) ->
    inv n (Node col a y b) (\{Y} \u A \u B).

Global Instance set_rep : Rep fset (set T).
Proof. apply (Build_Rep (fun e E =>
  exists n, inv n e E /\ root_color e = Black)).  [...]
Defined.
```

Figure 5. Representation predicate for red-black trees

```
Lemma insert_spec : RepTotal insert (X;elem) (E;fset) >>
                      = \{X} \u E ; fset.
Proof.
  xcf. introv RepX (n&InvE&HeB).
  xfun_induction_nointro_on size (Spec ins e |R>>
      forall n E, inv true n e E -> R (fun e' =>
       inv (is_black (root_color e)) n e' (\{X} \u E))).
    clears s n E. intros e IH n E InvE. inverts InvE as.
    xgo*. simpl. constructors*.
    introv InvA InvB RepY GtY LtY Col Num. xgo~.
    (* case insert left *)
    destruct~ col; destruct (root_color a); tryifalse~.
    ximpl as e. simpl. applys_eq* Hx 1 3.
    (* case insert right *)
    destruct~ col; destruct (root_color b); tryifalse~.
    ximpl as e. simpl. applys_eq* Hx 1 3.
    (* case no insertion *)
    asserts_rewrite~ (X = Y). apply~ nlt_nslt_to_eq.
      subst s. simpl. destruct col; constructors*.
  xlet as r. xapp~. inverts Pr; xgo. fset_inv. exists*.
Qed.
```

Figure 6. Invariant and model of red-black trees

but note that OCaml is a concise language and that Okasaki's code is particularly minimalist. Details are given about Coq scripts.

The column "inv" indicates the number of lines needed to state the invariant of each structure. The column "facts" gives the length of proof script needed to state and prove facts that are used several times in the verification scripts. The column "spec" indicates the number of lines of specification involved, including the specification of local and auxiliary functions. Finally, the last column describes the size of the actual verification proof scripts where characteristic formulae are manipulated. Note that Coq proof scripts also contain several lines to import and instantiate modules, a few lines to set up automation, as well as one line per function to register its specification in a database of lemmas.

We evaluate the relative cost of a formal verification by comparing the number of lines specific to formal proofs (figures from columns "facts" and "verif") against the number of lines required in a properly-documented source code (source code plus invariants and specifications). For particularly-tricky data structures, such as bootstrapped queues, Hood-Melville queues and binominal heaps, this ratio is close to 2.0. In all other structures, the ration does not exceed 1.25. For a user as fluent in Coq proofs as in OCaml programming, it means that the formalization effort can be expected to be comparable to the implementation and documentation effort.

Development	Caml	Coq	inv	facts	spec	verif
BatchedQueue	20	73	4	0	16	16
BankersQueue	19	95	6	20	15	16
PhysicistsQueue	28	109	8	10	19	32
RealTimeQueue	26	104	4	12	21	28
ImplicitQueue	35	149	25	21	14	50
BootstrappedQueue	38	212	22	54	29	77
HoodMelvilleQueue	41	363	43	53	33	180
BankersDeque	46	172	7	26	24	58
LeftistHeap	36	132	16	28	15	22
PairingHeap	33	137	13	17	16	35
LazyPairingHeap	34	132	12	24	14	32
SplayHeap	53	176	10	41	20	59
BinomialHeap	48	367	24	118	41	110
UnbalancedSet	21	85	9	11	5	22
RedBlackSet	35	183	20	43	22	53
BottomUpMergeSort	29	151	23	31	9	40
CatenableList	38	153	9	20	23	37
RandomAccessList	63	272	29	37	47	83
Total	643	3065	284	566	383	950

Figure 7. Non-empty lines of source code and proof scripts

4. Characteristic formula generation

4.1 Source language and normalization

CFML takes as input programs written in the pure fragment of OCaml, which includes algebraic data types, pattern matching, higher-order functions, recursion and mutual recursion. Polymorphic recursion, whose support was recently added to OCaml and which is used extensively in Okasaki's book, is also handled. Modules and functors are supported as long as the corresponding signatures are definable in Coq's module system.

Lazy expressions are supported under the condition that the code would terminate without any lazy annotation. While this restriction certainly does not enable reasoning on infinite data structures, it covers the use of laziness for computation *scheduling*, as described in Okasaki's book. In fact, our tools simply ignores any annotation relative to laziness. The key idea is that if a program satisfies its specification when evaluated without any lazy annotation, then it also satisfies its specification when evaluated with lazy annotations. (Of course, the reciprocal is not true.)

Program verification based on characteristic formulae could presumably be applied to another programming language. Yet, we make the assumption throughout this work that the source language is call-by-value and deterministic. For the sake of simplicity, program integers are modelled as unbounded mathematical integers.

Before generating the characteristic formula of a program, the program is automatically transformed into its *normal form*: the program is arranged so that all intermediate results and all functions become bound by a let-definition (except applications of simple total functions such as addition and subtraction). This transformation, similar to A-normalization [8], is straightforward to implement and greatly simplifies formal reasoning on programs (see [10, 24] for similar transformations in the context of program verification). The grammar of terms in normal form is given below, for a subset of the source language. It will later be extended with curried n-ary functions and curried n-ary applications (§5.3).

$$\begin{array}{lcl} x, f & := & \textit{variables} \\ v & := & x \mid n \mid (v, v) \mid \mathsf{inj}^k\, v \\ t & := & v \mid (v\, v) \mid \mathsf{fail} \mid \mathsf{if}\, x\, \mathsf{then}\, t\, \mathsf{else}\, t \mid \\ & & \mathsf{let}\, x = t\, \mathsf{in}\, t \mid \mathsf{let}\, f = (\mu f.\lambda x.t)\, \mathsf{in}\, t \end{array}$$

Throughout this work, we consider only programs that are well-typed in ML with recursive types. The grammar of types and type schema is recalled below.

$$\begin{array}{lcl} T & := & A \mid \mathsf{int} \mid T \times T \mid T + T \mid T \to T \mid \mu A.T \\ S & := & \forall \overline{A}.T \end{array}$$

4.2 Characteristic formula generation: informal presentation

The characteristic formula of a term t, written $[\![t]\!]$, is generated using a recursive algorithm that follows the structure of t. Recall that, given a post-condition P, the characteristic formula is such that the proposition "$[\![t]\!]\, P$" holds if and only if the term t terminates and returns a value that satisfies P. In terms of a denotational interpretation, $[\![t]\!]$ corresponds to the set of post-conditions that are valid for the term t. In terms of types, the characteristic formula associated with a term t of type T applies to a post-condition P of type $T \to \mathsf{Prop}$ and produces a proposition, so $[\![t]\!]$ admits the type $(T \to \mathsf{Prop}) \to \mathsf{Prop}$.

The key ideas involved in the construction of characteristic formulae are explained next. The reflection of Caml values into Coq and the treatment of polymorphism are described afterwards. The definition of $[\![t]\!]$ for a particular term t always takes the form "$\lambda P.\, H$", where H expresses what needs to be proved in order to show that the term t returns a value satisfying the post-condition P.

To show that a value v returns a value satisfying P, it suffices to prove that "$P\, v$" holds. So, $[\![v]\!]$ is defined as "$\lambda P.\ (P\ v)$". Next, to prove that an application "$f\, v$" returns a value satisfying P, one must exhibit a proof of "$\mathsf{AppReturns}\, f\, v\, P$". So, $[\![f\, v]\!]$ is defined as "$\lambda P.\ \mathsf{AppReturns}\, f\, v\, P$". To show that "$\mathsf{if}\, x\, \mathsf{then}\, t_1\, \mathsf{else}\, t_2$" returns a value satisfying P, one must prove that t_1 returns such a value when x is true and that t_2 returns such a value when x is false. So, the formula $[\![\mathsf{if}\, x\, \mathsf{then}\, t_1\, \mathsf{else}\, t_2]\!]$ is defined as

$$\lambda P.\ (x = \mathsf{true} \ \Rightarrow\ [\![t_1]\!]\, P) \ \wedge\ (x = \mathsf{false} \ \Rightarrow\ [\![t_2]\!]\, P)$$

To show that the term "fail" returns a value satisfying P, the only way to proceed is to show that this point of the program cannot be reached, by proving that the assumptions accumulated at that point are contradictory. Therefore, $[\![\mathsf{fail}]\!]$ is defined as "$\lambda P.\ \mathsf{False}$".

The treatment of let-bindings is more interesting. To show that a term "$\mathsf{let}\, x\ =\ t_1\, \mathsf{in}\, t_2$" returns a value satisfying P, one must prove that there exists a post-condition P' such that t_1 returns a value satisfying P' and that t_2 returns a value satisfying P for any x satisfying P'. Formally, $[\![\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2]\!]$ is defined as

$$\lambda P.\ \exists P'.\ ([\![t_1]\!]\, P') \ \wedge\ \forall x.\ (P'\ x) \ \Rightarrow\ ([\![t_2]\!]\, P)$$

Slightly trickier is the treatment of functions and recursive functions. In fact, we generate the same formula regardless of whether a function is recursive or not (except, of course, for the treatment of binding scopes). Indeed, as suggested in the example of the function half (§1.2), specification for recursive functions are proved by induction, using the induction principles provided by Coq. Thus, there is no need to add further support for reasoning by induction inside characteristic formulae.

Consider a possibly-recursive function "$\mu f.\lambda x.t$". The statement "$\forall x.\, \forall P'.\, [\![t]\!]\, P' \ \Rightarrow\ \mathsf{AppReturns}\, f\, x\, P'$", called the *body description* for f, captures the fact that, in order to prove that the application of f to x returns a value satisfying a post-condition P', it suffices to prove that the body t, instantiated with that particular value of x, terminates and returns a value satisfying P'. The characteristic formula for the function $\mu f.\lambda x.t$ then states that, in order to prove a property P to hold of $\mu f.\lambda x.t$, it suffices to prove that the *body description* for f implies the proposition "$P\, f$" for any abstract name f. The formula $[\![\mu f.\lambda x.t]\!]$ is thus defined as:

$$\lambda P.\ \forall f.\ \big(\forall x.\ \forall P'.\ [\![t]\!]\, P' \ \Rightarrow\ \mathsf{AppReturns}\, f\, x\, P'\big) \ \Rightarrow\ P\, f$$

The treatment of pattern matching and mutually-recursive functions can be found in the technical appendix [3].

4.3 Reflection of values in the logic

So far, we have abusively identified program values from the programming language with values from the logic. This section clarifies the translation from ML types to Coq types, as well as the translation from ML values to Coq values.

We map every ML value to its corresponding Coq value, except for functions. As explained earlier on, due to the mismatch between the programming language arrow type and the logical arrow type, we represent OCaml functions using values of type Func. For each ML type T, we define the corresponding Coq type, written $\langle T \rangle$. This type is simply a copy of T where all the arrow types are replaced with the type Func. Formally:

$$\begin{array}{lcl} \langle A \rangle & \equiv & A \\ \langle \mathsf{int} \rangle & \equiv & \mathsf{Int} \\ \langle T_1 \times T_2 \rangle & \equiv & \langle T_1 \rangle \times \langle T_2 \rangle \\ \langle T_1 + T_2 \rangle & \equiv & \langle T_1 \rangle + \langle T_2 \rangle \\ \langle \mu A.T \rangle & \equiv & \mu A.\langle T \rangle \\ \langle T_1 \to T_2 \rangle & \equiv & \mathsf{Func} \end{array}$$

Technical remark: a ML algebraic data type definition can be translated into a Coq inductive definition without any difficulty regarding negative occurrences. Indeed, since all arrow types are mapped to Func, there simply cannot be any negative occurrence.

Now, given a type T, we define the translation from Caml values of type T towards Coq values of type $\langle T \rangle$. The translation of a value v of type T is written $\lceil v \rceil^{\Gamma}_{T}$. The context Γ, which maps Caml variables to Coq variables, is used to translate non-closed values. The definition of the operator $\lceil \cdot \rceil$, called *decoder*, appears next.

$$\begin{array}{lcl} \lceil x \rceil^{\Gamma}_{T} & \equiv & \Gamma(x) \\ \lceil n \rceil^{\Gamma}_{\mathsf{int}} & \equiv & n \\ \lceil (v_1, v_2) \rceil^{\Gamma}_{T_1 \times T_2} & \equiv & (\lceil v_1 \rceil^{\Gamma}_{T_1}, \lceil v_2 \rceil^{\Gamma}_{T_2}) \\ \lceil \mathsf{inj}^k\, v \rceil^{\Gamma}_{T_1 + T_2} & \equiv & \mathsf{inj}^k\, \lceil v \rceil^{\Gamma}_{T_k} \\ \lceil v \rceil^{\Gamma}_{\mu A.T} & \equiv & \lceil v \rceil^{\Gamma}_{([A \to (\mu A.T)]\, T)} \\ \lceil \mu f.\lambda x.t \rceil^{\Gamma}_{T_1 \to T_2} & \equiv & \textit{not needed at this time} \end{array}$$

When decoding closed values, the context Γ is typically empty. Henceforth, we write $\lceil v \rceil_T$ as a shorthand for $\lceil v \rceil^{\emptyset}_{T}$. Moreover, when there is no ambiguity on the type T of the value v, we omit the type T and simply write $\lceil v \rceil^{\Gamma}$ and $\lceil v \rceil$.

4.4 Characteristic formula generation: formal presentation

The characteristic formula generator can now be given a formal presentation in which OCaml values are reflected into Coq, through calls to the decoding function $\lceil \cdot \rceil$. If t is a term of type T, then its characteristic formula $[\![t]\!]^{\Gamma}$ is actually a logical predicate of type $(\langle T \rangle \to \mathsf{Prop}) \to \mathsf{Prop}$. The environment Γ describes the substitution from program variables to Coq variables.

In order to justify that characteristic formulae can be displayed like the source code, we proceed in two steps. First, we describe the characteristic formula generator in terms of an intermediate layer of notation (Figure 8). Then, we define the notation layer in terms of higher-order logic connectives as well as in terms of the predicate AppReturns (Figure 9). The contents of those figures simply refines the informal presentation from §4.2.

4.5 Polymorphism

The treatment of polymorphism is certainly one of the most delicate aspect of characteristic formula generation. We need to extend the characteristic formula so as to quantify type variables needed to type-check the bodies of polymorphic let-bindings.

The translation of a polymorphic OCaml type $\forall \overline{B}.T$ is a polymorphic Coq type of the form $\forall \overline{A}.\langle T \rangle$. The set of type variables $\overline{A}$ is obtained by removing from the set $\overline{B}$ all the type variables that do not occur free in $\langle T \rangle$. Indeed, as all arrow types are mapped directly towards the type Func, some variables occuring in T may no longer occur in $\langle T \rangle$. So, the set $\overline{B}$ might be strictly smaller than $\overline{A}$.

Consider a polymorphic let-binding "$\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2$". The type checking of the term t_1 involves a set of type variables that are to be generalized at this let-binding on variable x. Let $\overline{C}$ denotes that set of generalizable type variables, and let T be the type of t_1 before generalization. The variable x thus admits a type of the form $\forall \overline{B}.T$, where $\overline{B}$ is a subset of $\overline{C}$. Note that, in general, $\overline{C}$ is a strict subset of $\overline{B}$ because not all intermediate type variables are visible in the result type of an expression.

Our goal is to define the characteristic formula associated with the term "$\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2$" in a context Γ. To that end, let $\forall \overline{A}.\langle T \rangle$ be the Coq translation of the type $\forall \overline{B}.T$. Since $\overline{A}$ is a subset of $\overline{B}$ and $\overline{B}$ is a subset of $\overline{C}$, we can define a set $\overline{A'}$ such that $\overline{C}$ is equal to the union of $\overline{A}$ and $\overline{A'}$. Then, we define:

$$\begin{array}{l} [\![\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2]\!]^{\Gamma} \equiv \\ \lambda P.\ \exists P' : (\forall \overline{A}.(\langle T \rangle \to \mathsf{Prop})).\ (\forall \overline{A}.\ \forall \overline{A'}.\ [\![t_1]\!]^{\Gamma}\ (P'\ \overline{A})) \\ \quad \wedge \forall X : (\forall \overline{A}.\langle T \rangle).\ (\forall \overline{A}.\ (P'\ \overline{A})\ (X\ \overline{A})) \Rightarrow ([\![t_2]\!]^{(\Gamma, x \mapsto X)}\ P) \end{array}$$

The post-condition P' describing X is a polymorphic predicate of type $\forall \overline{A}.(\langle T \rangle \to \mathsf{Prop})$. Note that it is not a predicate on a polymorphic value, which would have the type $(\forall \overline{A}.\langle T \rangle) \to \mathsf{Prop}$. (Indeed, we only care about describing the behaviour of monomorphic instances of the polymorphic variable X.) If we write type applications explicitly, then a particular monomorphic instance of X takes the form $X\ \overline{A}$ and it satisfies the predicate $P'\ \overline{A}$. Those type applications appear in the characteristic formula stated above.

Remark: we need to update slightly the translation from OCaml variables to Coq variables, because the context Γ may now associate program variables with *polymorphic* logical variables. The translation a monomorphic occurrence of a polymorphic variable x is the application of the Coq type variable $\Gamma(x)$ to some appropriates types, which depend on the type of x at its place of occurrence.

Finally, we give the characteristic formula for polymorphic functions, which is simpler than that of other polymorphic values because functions are simply reflected in the logic using the type Func. If $\overline{A}$ denotes the set of generalizable type variables associated with the body t of a function $\mu f.\lambda x.t$, then the characteristic formula is constructed as follows.

$$\begin{array}{l} [\![\mu f.\lambda x.t]\!]^{\Gamma} \equiv \lambda P.\ \forall F. \\ (\forall \overline{A}\ X\ P'.\ [\![t]\!]^{(\Gamma, f \mapsto F, x \mapsto X)}\ P' \Rightarrow \mathsf{AppReturns}\ F\ X\ P') \Rightarrow P\ F \end{array}$$

5. Specification predicates

Through this section, we formally describe the meaning of the predicates AppReturns and Spec. We then generalize those predicates to n-ary functions. Finally, we investigate how the predicate Spec can be used to specify higher-order functions.

5.1 Definition of the specification predicate

Consider the specification of the function half, written in terms of the predicate AppReturns.

$$\forall x.\ \forall n \geq 0.\ x = 2 * n \Rightarrow \mathsf{AppReturns}\ \mathsf{half}\ x\ (= n)$$

The same specification can be rewritten with the Spec notation as:

$$\mathsf{Spec}\ \mathsf{half}\ (x : \mathsf{int}) \mid R \gg \forall n \geq 0.\ x = 2 * n \Rightarrow R\ (= n)$$

The notation based on Spec in fact stands for an application of a higher-order predicate called Spec_1. The proposition "$\mathsf{Spec}_1\ f\ K$" asserts that the function f admits the specification K. The predicate K takes both x and R as argument, and specifies the result of the application of f to x. The predicate R is to be applied to the

$$
\begin{array}{lcl}
[\![v]\!]^\Gamma & \equiv & \mathsf{Ret}\ \lceil v \rceil^\Gamma \\
[\![f\ v]\!]^\Gamma & \equiv & \mathsf{App}\ \lceil f \rceil^\Gamma\ \lceil v \rceil^\Gamma \\
[\![\mathsf{fail}]\!]^\Gamma & \equiv & \mathsf{Fail} \\
[\![\mathsf{if}\ x\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2]\!]^\Gamma & \equiv & \mathsf{If}\ \lceil x \rceil^\Gamma\ \mathsf{Then}\ [\![t_1]\!]^\Gamma\ \mathsf{Else}\ [\![t_2]\!]^\Gamma \\
[\![\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2]\!]^\Gamma & \equiv & \mathsf{Let}\ X\ :=\ [\![t_1]\!]^\Gamma\ \mathsf{in}\ [\![t_2]\!]^{(\Gamma, x \mapsto X)} \\
[\![\mathsf{let}\ f' = (\mu f. \lambda x. t_1)\ \mathsf{in}\ t_2]\!]^\Gamma & \equiv & \mathsf{Let}\ F'\ :=\ \big(\mathsf{Fun}\ F\ X\ :=\ [\![t_1]\!]^{(\Gamma, f \mapsto F, x \mapsto X)}\big)\ \mathsf{in}\ [\![t_2]\!]^{(\Gamma, f' \mapsto F')}
\end{array}
$$

Figure 8. Characteristic formula generator

$$
\begin{array}{lcl}
\mathsf{Ret}\ V & \equiv & \lambda P.\ P\,V \\
\mathsf{App}\ F\ V & \equiv & \lambda P.\ \mathsf{AppReturns}\,F\,V\,P \\
\mathsf{Fail} & \equiv & \lambda P.\ \mathsf{False} \\
\mathsf{If}\ V\ \mathsf{Then}\ Q\ \mathsf{Else}\ Q' & \equiv & \lambda P.\ (V = \mathsf{true} \Rightarrow Q\,P) \wedge (V = \mathsf{false} \Rightarrow Q'\,P) \\
\mathsf{Let}\ X\ :=\ Q\ \mathsf{in}\ Q' & \equiv & \lambda P.\ \exists P'.\ Q\,P' \wedge (\forall X.\ P'\,X \Rightarrow Q'\,P) \\
\mathsf{Fun}\ F\ X\ :=\ Q & \equiv & \lambda P.\ \forall F.\ \big(\forall X.\ \forall P'.\ Q\,P' \Rightarrow \mathsf{AppReturns}\,F\,X\,P'\big) \Rightarrow P\,F
\end{array}
$$

Figure 9. Syntactic sugar to display characteristic formulae

post-condition that holds of the result of "$f\,x$". For example, the previous specification for half stands for:

$$\mathsf{Spec}_1\ \mathsf{half}\ (\lambda x\,R.\ \forall n \geq 0.\ x = 2 * n \Rightarrow R\,(= n))$$

In first approximation, the predicate Spec_1 is defined as follows:

$$\mathsf{Spec}_1\,f\,K \quad \equiv \quad \forall x.\ K\,x\,(\mathsf{AppReturns}\,f\,x)$$

where K has type $A \to ((B \to \mathsf{Prop}) \to \mathsf{Prop}) \to \mathsf{Prop}$, where A and B correspond to the input and the output type of f, respectively. The reader may check that unfolding the definition of Spec_1 in the specification for half expressed using Spec_1 yields the specification for half expressed in terms of AppReturns.

The true definition of "Spec_1" actually include an extra side-condition, expressing that K is covariant in R. It is needed to ensure that the specification K actually concludes about the behaviour of the application of the function. Formally, covariance is captured by the predicate Weakenable, defined as follows:

$$\mathsf{Weakenable}\,H \ \equiv\ \forall\,G\,G'.\ (\forall x.\,G\,x \to G'\,x) \to H\,G \to H\,G'$$

where H has type "$(X \to \mathsf{Prop}) \to \mathsf{Prop}$" for some X. The formal definition of Spec_1 appears in the middle of Figure 10. Fortunately, thanks to appropriate lemmas and tactics, the predicate Weakenable never needs to be manipulated explicitly by the user.

5.2 Direct treatment of n-ary functions

In order to obtain a realistic tool for program verification, it is crucial to offer direct support for reasoning on the definition and application of n-ary curried functions. Generalizing the definitions of Spec_1 and $\mathsf{AppReturns}_1$ to higher arities is not entirely straightforward, because we want the ability to reason on partial applications and over applications. Intuitively, the specification of a n-ary curried function should capture the property that the application to a number of arguments less than n terminates and returns a function with the appropriate specialization of the original specification.

Firstly, we define the predicate $\mathsf{AppReturns}_n$. The proposition "$\mathsf{AppReturns}_n\,f\,v_1 \ldots v_n\,P$" states that the application of f to the n arguments $v_1 \ldots v_n$ returns a value satisfying P. The family of predicates $\mathsf{AppReturns}_n$ is defined by recursion on n in terms of the predicate AppReturns, as shown at the top of Figure 10. For instance, "$\mathsf{AppReturns}_2\,f\,v_1\,v_2\,P$" states that the application of f to v_1 returns a function g such that the application of g to v_2 returns a value satisfying P. More generally, if m is smaller than n, then applications at arities n and m are related as follows:

$$\begin{array}{l}\mathsf{AppReturns}_n\,f\,v_1 \ldots v_n\,P \quad \Longleftrightarrow \\ \mathsf{AppReturns}_m\,f\,v_1 \ldots v_m\,(\lambda g.\,\mathsf{AppReturns}_{n-m}\,g\,v_{m+1} \ldots v_n\,P)\end{array}$$

Secondly, we define the predicate Spec_n. Again, we proceed by recursion on n. For example, a curried function f of two arguments is a total function that, when applied to its first argument, returns a unary function g that admits a certain specification which depends on that first argument. Formally:

$$\mathsf{Spec}_2\ f\ K \quad \equiv \quad \mathsf{Spec}_1\ f\ (\lambda x\,R.\ R\ (\lambda g.\,\mathsf{Spec}_1\,g\,(K\,x)))$$

where $(K : A_1 \to A_2 \to ((B \to \mathsf{Prop}) \to \mathsf{Prop}) \to \mathsf{Prop})$. Remark: Spec_2 is polymorphic in the types A_1, A_2 and B.

The actual definition, given in Figure 10, includes a side condition to ensure that K is covariant in R, written $\mathsf{Is\_spec}_n\,K$. Note: the specification of a curried function described using Spec_n can always be viewed as a unary function specified using Spec_1. This property will be useful for reasoning on higher-order functions.

The high-level notation for specification used in §3 can now be easily explained in terms of the family of predicates Spec_n.

$$\begin{array}{l}\mathsf{Spec}\ f\ (x_1 : A_1) \ldots (x_n : A_n)\ |\ R \gg H \\ \equiv\ \ \mathsf{Spec}_n\ f\ (\lambda(x_1 : A_1).\ \ldots \lambda(x_n : A_1).\,\lambda R.\ H)\end{array}$$

5.3 Characteristic formulae for curried functions

In this section, we update the generation of characteristic formulae to add direct support for reasoning on n-ary functions using Spec_n and $\mathsf{AppReturns}_n$. Note that the grammar of terms in normal form is now extended with n-ary applications and n-ary abstractions.

Intuitively, the characteristic formula associated with an application "$f\,v_1 \ldots v_n$" is simply "$\lambda P.\ \mathsf{AppReturns}_n\,v_1 \ldots v_n\,P$". The formal definition, which takes decoders into account, is:

$$[\![f\,v_1 \ldots v_n]\!]^\Gamma \equiv\ \lambda P.\ \mathsf{AppReturns}_n\,\lceil f \rceil^\Gamma\,\lceil v_1 \rceil^\Gamma \ldots \lceil v_n \rceil^\Gamma\,P$$

The characteristic formula for a function "$\mu f.\lambda x_1 \ldots x_n.t$" asserts that to prove "$\mathsf{Spec}_n\,f\,K$" it suffices to show that the proposition "$K\,x_1 \ldots x_n\,[\![t]\!]$" holds for any arguments x_i. Remark: the treatment of unary functions given here is different but provably equivalent to that given earlier on (§4.2).

It may be surprizing to see the predicate "$K\,x_1 \ldots x_n$" being applied to a characteristic formula $[\![t]\!]$. It is worth considering an example. Recall the definition of the function half. It takes the form "$\mu\mathsf{half}.\lambda x.t$", where t stands for the body of half. Its specification takes the form "$\mathsf{Spec}_1\,\mathsf{half}\,K$", where K is equal to "$\lambda x\,R.\ \forall n \geq 0.\ x = 2 * n \Rightarrow R\,(= n)$". According to the new characteristic formula for functions, in order to prove that the function half satisfies its specification, we need to prove the proposition "$\forall x.\ K\,x\,[\![t]\!]$". Unfolding K, we obtain: "$\forall n \geq 0.\ x = 2 * n \Rightarrow [\![t]\!]\,(= n)$". As expected, we are required to prove that the body of the function half (described by the characteristic formula

$$
\begin{array}{lcl}
\mathsf{AppReturns}_1\ f\ x\ P & \equiv & \mathsf{AppReturns}\ f\ x\ P \\
\mathsf{AppReturns}_n\ f\ x_1 \ldots x_n\ P & \equiv & \mathsf{AppReturns}\ f\ x_1\ (\lambda g.\ \mathsf{AppReturns}_{n-1}\ g\ x_2 \ldots x_n\ P) \\
\mathsf{Is\_spec}_1\ K & \equiv & \forall x.\ \mathsf{Weakenable}\ (K\ x) \\
\mathsf{Is\_spec}_n\ K & \equiv & \forall x.\ \mathsf{Is\_spec}_{n-1}\ (K\ x) \\
\mathsf{Spec}_1\ f\ K & \equiv & \mathsf{Is\_spec}_1\ K\ \wedge\ \forall x.\ K\ x\ (\mathsf{AppReturns}\ f\ x) \\
\mathsf{Spec}_n\ f\ K & \equiv & \mathsf{Is\_spec}_n\ K\ \wedge\ \mathsf{Spec}_1\ f\ (\lambda x\ R.\ R\ (\lambda g.\ \mathsf{Spec}_{n-1}\ g\ (K\ x)))
\end{array}
$$

In the figure, $n > 1$ and $(f : \mathsf{Func})$ and $(x_i : A_i)$ and $(P : B \to \mathsf{Prop})$ and $(K : A_1 \to \ldots A_n \to ((B \to \mathsf{Prop}) \to \mathsf{Prop}) \to \mathsf{Prop})$.

Figure 10. Formal definitions for $\mathsf{AppReturns}_n$ and Spec_n

$[\![t]\!]$) returns a value equal to n, under the assumption that n is a non-negative integer such that $x = 2 * n$.

Characteristic formulae for functions are constructed as follows.

$$
[\![\mu f.\lambda x_1 \ldots x_n.t]\!]^\Gamma \equiv \lambda P.\ \forall F.\ \left(\forall K.\ \left(\forall X_1 \ldots X_n.\ K\ X_1 \ldots X_n\ [\![t]\!]^{(\Gamma, f \mapsto F, x_i \mapsto X_i)}\right) \Rightarrow \mathsf{Is\_spec}_n\ K\ \Rightarrow\ \mathsf{Spec}_n\ F\ K\right) \Rightarrow P\ F
$$

5.4 Specification of higher-order functions

The specification of a function, whether unary or n-ary, can always take the form $\mathsf{Spec}_1\ f\ K$. Thus, given a function f, we can quantify over every possible specification that f might admit simply by quantifying universally over the variable K. Let us illustrate this ability with the functions apply and compose. The function apply, defined as "$\lambda x.\ \lambda f.\ (f\ x)$", can be specified as follows.

$$
\mathsf{Spec}_2\ \mathsf{apply}\ (\lambda x\ f\ R.\ \forall K.\ \mathsf{Spec}_1\ f\ K\ \Rightarrow\ K\ x\ R)
$$

The conclusion "$K\ x\ R$" states that the behaviour R of the term "apply $f\ x$" is described by the predicate "$K\ x$". The predicate "$K\ x$" indeed specifies the behaviour of the term "$f\ x$", since "$\mathsf{Spec}_1\ f\ K$" implies "$K\ x\ (\mathsf{AppReturns}_1\ f\ x)$".

Consider now the function compose, which is defined as "$\lambda f_1\ f_2\ x.\ f_1\ (f_2\ x)$". Its specification is expressed in terms of the specifications K_1 and K_2 of the functions f_1 and f_2, respectively.

$$
\begin{array}{l}
\mathsf{Spec}_3\ \mathsf{compose}\ (\lambda f_1\ f_2\ x\ R. \\
\quad \forall K_1\ K_2.\ \mathsf{Spec}_1\ f_1\ K_1\ \Rightarrow\ \mathsf{Spec}_1\ f_2\ K_2\ \Rightarrow \\
\quad K_2\ x\ (\lambda P.\ \exists y.\ P\ y\ \Rightarrow\ K_1\ y\ R)))
\end{array}
$$

The last line can be read as follows. First, we want to unfold the specification "$K_2\ x$" associated with the application of f_2 onto x, since this inner call is the first to be performed. Then, for any post-condition P that holds of the result y of the application "$f_2\ x$", the behaviour R of the term "$f_1\ (f_2\ x)$" is the same as the behaviour of "$f_1\ y$". Since the behaviour of "$f_1\ y$" is described by the predicate "$K_1\ y$", the conclusion is "$K_1\ y\ R$".

The specification given above specifies in particular the result obtained by applying compose to two functions. For example, we were able to prove in a few lines of Coq that the term "compose half half" yields a function that divides its argument by four. More precisely, using a weakening lemma for specification, we have proved that the resulting function admits the specification "$\lambda x\ R.\ \forall n \geq 0.\ x = 4 * n\ \Rightarrow\ R\ (= n)$". (See [3] for details.)

Using similar techniques, we were able to assign a concise specification to the Y fixed-point combinator, and then to verify it. We have also started to investigate the specification of higher-order iterators such as map and fold on lists and sets. However, due to lack of space and because we lack experience in *using* those specifications, we do not report on that recent work in this paper.

6. Soundness and completeness

Characteristic formulae can be displayed in a way that closely resemble source code. However, proving the soundness and completeness of a characteristic formula with respect to the source code it describes is not entirely straightforward. First, we show how the type Func and the predicate AppReturns can be given concrete implementations in the logic. This construction, which has been verified in Coq for a subset of the source language, relies on a deep embedding of the source language and on the definition of functions called *encoders*, which are the reciprocal of *decoders*. Second, we present the statements of the soundness and completeness theorems, which have been proved on paper [3].

6.1 Realization of Func and AppReturns

To realize the type Func, we construct a deep embedding of the source language. More precisely, we use inductive definitions to define the set of runtime values, named Val, and to define the set of program terms, named Trm. Runtime values, written v throughout this section, extend source program values with function closures. We then define Func as the set of function closures, that is, as the set of values of type Val of the form $\mu f.\lambda x.t$. In order to prove interesting facts about characteristic formulae, we need to define a decoder for function closures created at runtime. We define the decoding of a function as the deep embedding of the code of that function. In other words, the decoder for functions is the identity.

$$
\lceil \mu f.\lambda x.t \rceil^\Gamma_{T_1 \to T_2} \quad \equiv \quad (\mu f.\lambda x.t) : \mathsf{Func}
$$

Note that the context Γ is ignored as function closures are always closed values.

To realize the predicate AppReturns, we need to define the semantics of the source language and to define *encoders*. First, we describe the semantics of the deep embedding of the source language through a big-step reduction relation. This inductively-defined judgment, written "$t \Downarrow v$", relates a term t of type Trm with a value v of type Val. Second, we define *encoders*, which are the reciprocal of decoders. For each program type T, we define an encoder function, written $\lfloor V \rfloor_{\langle T \rangle}$ or simply $\lfloor V \rfloor$, that translates a logical value V of type $\langle T \rangle$ towards the deep embedding of the corresponding program value. Thus, $\lfloor V \rfloor_{\langle T \rangle}$ is always a logical value of type Val. The definition of encoders, not shown here, is such that $\lfloor \lceil v \rceil_T \rfloor_{\langle T \rangle} = v$ and $\lceil \lfloor V \rfloor_{\langle T \rangle} \rceil_T = V$.

We can now give the concrete implementation to AppReturns. The judgment "$\mathsf{AppReturns}\ F\ V\ P$" asserts that the application of F to the embedding of V terminates and returns the embedding of a value V' that satisfies P. Remark: since F is a value of type Func, F is also equal to its encoding $\lfloor F \rfloor$.

$$
\mathsf{AppReturns}\ F\ V\ P \quad \equiv \quad \exists V'.\ (P\ V')\ \wedge\ (F\ \lfloor V \rfloor) \Downarrow \lfloor V' \rfloor
$$

6.2 Soundness and completeness theorems

The soundness theorem states that if a predicate P satisfies the characteristic formula of a term t, then the term t terminates and returns the encoding of a value V satisfying P.

Theorem 6.1 (Soundness) *For any closed term t of type T and any predicate P of type "$\langle T \rangle \to \mathsf{Prop}$",*

$$
[\![t]\!]^\emptyset\ P \qquad \Rightarrow \qquad \exists V.\quad t \Downarrow \lfloor V \rfloor \quad \wedge \quad P\ V
$$

The completeness result states that the characteristic formula of a term implies any true specification satisfied by this term. To

avoid complications related to the occurrence of functions in the final result of a program, we present here only the particular case where the program produces an integer value as final result.

Theorem 6.2 (Completeness for integer results) *Let t be a well-typed closed term, n be an integer, and P be a predicate on integers. If "$t \Downarrow \lfloor n \rfloor$" and "$P\,n$" are true then the proposition "$[\![t]\!]^{\emptyset}\ P$" is provable, even without knowledge of the concrete definitions of* Func *and* AppReturns*.*

A more precise theorem can be found in the appendix [3].

6.3 Quantification over type variables

Polymorphism has been treated by quantifying over logical type variables, but we have not mentioned what exactly is the sort of these variables in the logic. A tempting solution would be to assign them the sort Type. (In Coq, Type is the sort of all types from the logic, including the sort of Prop.) But in fact, type variables used to represent ML polymorphism are only meant to range over reflected types, i.e. types of the form $\langle T \rangle$. Thus, we ought to assign type variables the sort RType, defined as $\{\, X : \mathsf{Type} \mid \exists T.\, X = \langle T \rangle \,\}$.

Since we provide RType as an abstract definition, users do not need to exploit the fact that universally-quantified types correspond to reflected ML types. A question naturally follows: since RType is an abstract type, would it remain sound and complete to use the sort Type instead of the sort RType as a sort for type variables? We conjecture that the answer is positive. In the implementation, we use the sort Type for the sake of convenience, however we could switch to RType if it ever turned out to be necessary.

7. Conclusion

We have presented CFML, a tool for the verification of pure OCaml programs. It consists of two parts: a characteristic formula generator (implemented in 3000 lines OCaml) and a set of lemmas, notation and tactics for manipulating characteristic formulae (a 4000-line Coq library). We have reused OCaml's parser and type-checker to achieve maximal compatibility, making it possible to verify existing code, even if it is was not originally intented to be verified.

We have employed our tool to specify and verify total correctness of a number of advanced purely-functional data structures. Complex invariants can be expressed concisely, thanks to the high expressiveness of higher-order logic. Nontrivial proof obligations can be discharged easily, thanks to the use of interactive proofs. When the code or its specification is incorrect, the proof assistant provides immediate feedback, explaining what proof obligation fails and where this obligation comes from. In our experience, the process of verifying a program can be conducted relatively quickly. Most often, the hardest part is to figure out very precisely all the invariants that the program relies upon.

References

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

[2] Arthur Charguéraud. Verification of call-by-value functional programs through a deep embedding. Unpublished. http://arthur.chargueraud.org/research/2009/deep/, March 2009.

[3] Arthur Charguéraud. Technical appendix to the current paper. http://arthur.chargueraud.org/research/2010/cfml/, April 2010.

[4] Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective interactive proofs for higher-order imperative programs. In *ICFP*, September 2009.

[5] Thierry Coquand. Alfa/agda. In Freek Wiedijk, editor, *The Seventeen Provers of the World*, volume 3600 of *Lecture Notes in Computer Science*, pages 50–54. Springer, 2006.

[6] Xinyu Feng, Zhong Shao, Alexander Vaynberg, Sen Xiang, and Zhaozhong Ni. Modular verification of assembly code with stack-based control abstractions. In M. Schwartzbach and T. Ball, editors, *PLDI*. ACM, 2006.

[7] 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.

[8] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *PLDI*, pages 237–247, 1993.

[9] G. A. Gorelick. A complete axiomatic system for proving assertions about recursive and non-recursive programs. Technical Report 75, University of Toronto, 1975.

[10] Kohei Honda, Martin Berger, and Nobuko Yoshida. Descriptive and relative completeness of logics for higher-order functions. In M. Bugliesi, B. Preneel, V. Sassone, and I. Wegener, editors, *ICALP (2)*, volume 4052 of *LNCS*. Springer, 2006.

[11] Johannes Kanig and Jean-Christophe Filliâtre. Who: a verifier for effectful higher-order programs. In *ML'09: Proceedings of the 2009 ACM SIGPLAN workshop on ML*, pages 39–48, New York, NY, USA, 2009. ACM.

[12] Henri Korver. Computing distinguishing formulas for branching bisimulation. In Kim Guldstrand Larsen and Arne Skou, editors, *CAV*, volume 575 of *LNCS*, pages 13–23. Springer, 1991.

[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 *LNCS*, pages 121–135. Springer, 2003.

[17] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[18] Magnus O. Myreen, Michael J. C. Gordon, and Konrad Slind. Machine-code verification for multiple architectures: an application of decompilation into logic. In *FMCAD*, pages 1–8, Piscataway, NJ, USA, 2008. IEEE Press.

[19] Aleksandar Nanevski, J. Gregory Morrisett, and Lars Birkedal. Hoare type theory, polymorphism and separation. *JFP*, 18(5-6):865–911, 2008.

[20] Aleksandar Nanevski, Viktor Vafeiadis, and Josh Berdine. Structuring the verification of heap-manipulating programs. In Manuel V. Hermenegildo and Jens Palsberg, editors, *POPL*, pages 261–274. ACM, 2010.

[21] Zhaozhong Ni and Zhong Shao. Certified assembly programming with embedded code pointers. In *POPL*, 2006.

[22] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.

[23] David Park. Concurrency and automata on infinite sequences. In Peter Deussen, editor, *Theoretical Computer Science: 5th GI-Conference, Karlsruhe*, volume 104 of *LNCS*, pages 167–183, Berlin, Heidelberg, and New York, March 1981. Springer-Verlag.

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

[25] Matthieu Sozeau. Program-ing finger trees in coq. *SIGPLAN Not.*, 42(9):13–24, 2007.

[26] Karen Zee, Viktor Kuncak, and Martin Rinard. An integrated proof language for imperative programs. In *PLDI*, 2009.