Characteristic Formulae for the Verification of Imperative Programs

Arthur Charguéraud

INRIA
arthur.chargueraud@inria.fr

Abstract. We have developed characteristic formulae as a technique for verifying imperative programs using interactive theorem provers. The characteristic formula of a program is a higher-order logic formula that gives a sound and complete description of the semantics of this program without referring to its source code. The formula can be constructed automatically from the source code it describes, in a compositional way. Moreover, it can be conveniently manipulated in interactive proofs. Characteristic formulae support reasoning about all forms of first-class functions, through the use of an abstract predicate used to specify functions extensionally in the logic. Moreover, they integrate techniques from Separation Logic in order to support modular reasoning about mutable data structures. Characteristic formulae serve as a basis for a tool, called CFML, which supports the verification of imperative Caml programs using the Coq proof assistant. Using CFML, we have formally verified nontrivial imperative algorithms, as well as CPS functions, higher-order iterators, and programs involving higher-order stores.

1 Introduction

In many applications, bugs can have dramatic consequences. Experience shows that it is not possible to find all the bugs of a program through intensive testing or static analysis. To ensure the complete absence of bugs, we need to prove that a program is correct with respect to a specification —the specification is assumed to be a valid formal description of the intended behavior of the program. The process of building a proof of correctness and having this proof be verified by a machine is called program verification. It allows to reach a very high degree of confidence in the correctness of the code. Yet, despite its importance, program verification is far from commonplace. Many approaches to program verification have been studied in the past 50 years, but none of them has yet succeeded in making program verification easily applicable to general-purpose programs.

The most common approach to program verification consists of annotating the program with its specification and its invariants, and then using a Verification Condition Generator (VCG) to extract a set of proof obligations. When these proof obligations are simple enough, SMT solvers can be used to discharge them automatically. This approach has so far shown to be quite effective for reasoning about simple programs. However, this approach shows its limits in the face of more complex programs. For example, when a program makes use of higher-order functions or requires inductive reasoning, SMT solvers are usually not able to make any progress. In theory, the user could use a proof assistant for discharging the proof obligations interactively. Yet, in practice, the proof obligations generated by a VCG, typically through the computation of a weakest precondition, are numerous and are not at all well-suited for human consumption.

A few researchers have developed interactive proof systems specifically for reasoning about programs (e.g., KeY [4]). However, the use of an ad-hoc logic precludes the use of standard proof assistants and its associated mathematical libraries. Other researchers have worked instead on trying to use a standard proof assistant as an environment in which to write programs and prove properties about them. This approach is known as *shallow embedding*. For example, Coq [12] contains a purely-functional language, and its extraction mechanism [28] allows to convert Coq definitions into Haskell or Caml code. However, there are fundamental discrepancies between a programming language, whose functions can be partial and perform side-effects, and a logical language, whose functions must be pure and total. The project Ynot [11, 37, 38] showed how to extend Coq with a monad for embedding side-effects and non-termination. This work has shown that a shallow embedding can be used to produce formally-verified imperative code. Yet, the language in which programs have to be written remains tightly constrained by Coq. Therefore, this approach cannot be used to verify existing programs written in a real programming language.

There exists a technique that, at least in theory, supports reasoning about any program written in any programming language. It consists of using a *deep embedding*, that is, a formalization of the syntax and the semantics of a programming language inside a proof assistant. Using the deep embedding of a given programming language, one can describe the source code of a program and prove any true property about its semantics (within the limits of the expressiveness of the theorem prover). However, in practice, working through a deep embedding is extremely cumbersome, because of the indirection that it involves. All the terms and values of the language must be encoded using the constructions of the deep embedding. In order to hide as much as possible the details of the embedding from the user, it is possible to set up a collection of notation and tactics. We have done so in our previous work on the deep embedding of purely-functional Caml programs in Coq [7]. Yet, the complexity of the deep embedding remains. In particular, specifications are polluted by the need to relate embedded program values with the corresponding Coq values.

We have developed characteristic formulae as a new, practical approach to program verification. The characteristic formula of a program is a logical formula that describes the semantics of this program, but without referring to a deep embedding of this program. This formula is built automatically from the source code alone and is expressed in a standard higher-order logic. To verify a program, the user states its specification in the form of a theorem and then prove this theorem using the characteristic formula. The proof follows the structure of the characteristic formula, which itself follows the structure of the code. Invariants involved in the proofs, such as loop invariants, are provided by the user during the interactive proofs.

Characteristic formulae can be used to prove that a program terminates and satisfies a particular specification. In fact, they can be used to prove any true property of the value returned by a program —we have indeed established a completeness result for characteristic formulae. In particular, the program does not need to be rewritten in a particular form: characteristic formulae can be applied to the verification of existing programs. Note that the notion of characteristic formula is not specific to a particular programming language. In this paper, we explain how to develop characteristic formulae for a large subset of Caml, and we demonstrate how these formulae can be used in practice to verify nontrivial programs.

The notion of characteristic formula originates in process calculi. In this context, two processes are behaviorally equivalent if and only if their characteristic formulae are logically equivalent [19]. Graf and Sifakis proposed in the 80's an algorithm for building the characteristic formula of any process [17]. More recently, Honda, Berger and Yoshida adapted this idea from process logics to program logics [21]. They gave an algorithm for building the pair of the weakest pre-condition and of the strongest post-condition of any PCF program. Note that their algorithm differs from weakest pre-condition calculus in that the PCF program considered is not assumed to be annotated with any invariant. Honda *et al* suggested that characteristic formulae could be used in program verification. However, they did not find a way to encode the ad-hoc logic that they were using for stating specifications into a standard logic. As a result, they could not reuse any existing theorem prover. Since the development of a dedicated theorem prover would have required too much effort, Honda *et al*'s work remained theoretical and did not result in an effective program verification tool.

We have developed characteristic formulae much further and have turned this concept into a practical approach to program verification. Our contribution is summarized next.

- We devise characteristic formulae that can be expressed in a standard higher-order logic, allowing for the use of an off-the-shelf proof assistants, e.g., Coq.
- Our characteristic formulae always have a size linear in that of the source code they describe. Verification using characteristic formulae therefore has a chance to scale up to large pieces of code.
- We show how to set up a notation system that allows to pretty-print a characteristic formula in a way that very closely resembles the source code that the formula describes. The notation system makes proof obligations easy to read and easy to relate to the source code.
- We explain how to integrate in characteristic formulae the frame rule from Separation Logic [43]. The frame rule allows for local reasoning and is essential to modularity.

- We have developed a set of Coq tactics that enables the user to efficiently manipulate characteristic formulae and conduct program verification without any knowledge of how characteristic formulae are constructed.
- We have implemented our approach in a tool, called CFML, which supports the verification of programs written in a large subset of Caml [27]. It targets the Coq proof assistant [12].
- We have applied CFML to the verification of a number of nontrivial imperative programs. In this paper, we report on the verification of Dijkstra's shortest path algorithm and on several examples illustrating nontrivial interactions between first-class functions and the mutable state.
- We have proved that our characteristic formulae are sound and complete. In this paper, we present the statement of the theorems and of the main invariants, however we do not include the proofs (they can be found in [8]).

The work described in the present paper corresponds to the core of the author's PhD thesis [8]. We presented characteristic formulae for purely-functional programs at ICFP'10 [9]. We later presented the generalization of characteristic formulae to imperative programs at ICFP'11 [10]. Compared with our ICFP'11 paper, the present paper contains more details. In particular, we describe characteristic formulae for assertions and pattern-matching, the treatment of n-ary functions, the properties of the star operator and the heap entailment relation. We also give additional details about the invariants of the soundness proof, and explain how to justify our treatment of polymorphism. The tool CFML, which directly implements all the ideas described in this paper, can be found online[1]. The CFML webpage also contains all the examples from this paper and additional examples, including those taken from the verification of more than half of the content of Okasaki's book on *Purely Functional Data Structures* [41].

The content of this paper is organized in four main parts. Section 2 describes the key ideas involved in the construction, the pretty-printing and the manipulation of characteristic formulae. Section 3 formalizes the translation of types and values, the representation and the specification of heaps, the heap entailment relation, the integration of the frame rule, and the construction of characteristic formulae. Section 4 presents the statement of the soundness and completeness theorems, and discusses the key invariants. Section 5 contains a presentation of several examples that we have specified and formalized using CFML. Section 6 discusses related work and Section 7 concludes.

2 Overview

2.1 Verification through characteristic formulae

The characteristic formula of a term t relates a description of the input heap in which the term t is executed with a description of the output value and a

[1] `http://arthur.chargueraud.org/softs/cfml/`

description of the output heap produced by the execution of t. Characteristic formulae are hence closely related to Hoare triples [20], and, more precisely, to *total correctness* Hoare triples, which account not just for correctness but also for termination.

A total correctness Hoare triple $\{H\}\ t\ \{Q\}$ asserts that, when executed in a heap satisfying the predicate H, the term t terminates and returns a value v in a heap satisfying $(Q\,v)$. The post-condition Q is thus used to specify both the output heap and the output value. When t has type τ, the pre-condition H has type $\mathsf{Heap} \to \mathsf{Prop}$ and the post-condition Q has type $\langle\tau\rangle \to \mathsf{Heap} \to \mathsf{Prop}$, where Heap is the type of a heap and where $\langle\tau\rangle$ is the Coq type that corresponds to the ML type τ. To clarify the role of post-conditions, we give two examples. The triple $\{r \hookrightarrow n\}\ (\mathsf{incr}\, r)\ \{\lambda(\_ : \mathsf{unit}).\ r \hookrightarrow n+1\}$ asserts that a call to the increment function on a reference r updates the content of this cell from n to $n+1$ and returns the unit value. The triple $\{[\,]\}\ (\mathsf{ref}\,3)\ \{\lambda r.\ r \hookrightarrow 3\}$ asserts that, starting from the empty heap, a call to the function ref with the argument 3 produces a heap made of a cell at a fresh location r whose content is the value 3.

The characteristic formula of a term t is written $[\![t]\!]$. This predicate is such that $[\![t]\!]\,H\,Q$ captures exactly the same proposition as the triple $\{H\}\ t\ \{Q\}$. There is however a fundamental difference between Hoare triples and characteristic formulae. A Hoare triple $\{H\}\ t\ \{Q\}$ is a three-place relation, whose second argument is a representation of the syntax of the term t. On the contrary, $[\![t]\!]\,H\,Q$ is a logical proposition, expressed in terms of standard higher-order logic connectives, such as $\wedge$, $\exists$, $\forall$ and $\Rightarrow$. Importantly, this proposition does not refer to the syntax of the term t. Moreover, whereas Hoare-triples need to be established by application of derivation rules specific to Hoare logic, characteristic formulae can be proved using only basic higher-order logic reasoning, without involving external derivation rules.

We have used characteristic formulae for building CFML, a tool that supports the verification of imperative Caml programs using the Coq proof assistant. CFML takes as input source code written in a large subset of Caml, and it produces as output a set of Coq axioms that correspond to the characteristic formulae of each top-level definition. It is worth noting that CFML generates characteristic formulae without knowledge of the specification nor of the invariants of the source code. The specification of each top-level definition is instead provided by the user, in the form of the statement of a Coq theorem. The user may prove such a theorem by exploiting the axiom generated by CFML for that definition. The user provides information such as loop invariants interactively during the proofs.

When reasoning about a program through its characteristic formula, a proof obligation typically takes the form $[\![t]\!]\,H\,Q$, asserting that the piece of code t admits H as pre-condition and Q as post-condition. The user can make progress in the proof by invoking the custom tactics provided by CFML. Proof obligations thereby get decomposed into simpler subgoals, following the structure of the code. When reaching a leaf of the source code, several facts may need to be established in order to justify the correctness of the program. These facts, which no longer contain any reference to characteristic formulae, can be proved using general-purpose Coq tactics, including calls to decision procedures and to proof-search algorithms.

The rest of this section presents the key ideas involved in the construction of characteristic formulae, covering the treatment of sequences, let bindings, the frame rule and functions.

2.2 Construction of characteristic formulae

Consider a sequence term, of the form "$t_1 \,;\, t_2$". The rule for reasoning on sequences in Hoare Logic (or Separation Logic) asserts that such a sequence admits a pre-condition H and a post-condition Q if there exists a heap predicate describing the state between the execution of t_1 and t_2. More precisely, assume that t_1 admits the pre-condition H and the post-condition Q'. The state produced by the execution of t_1 is described as $Q'\,tt$, where tt denotes the unit value produced by t_1. Thus, t_2 should admit the pre-condition $Q'tt$ and the post-condition Q. The reasoning rule for sequences is formally stated as follows.

$$\frac{\{H\}\; t_1\; \{Q'\} \qquad \{Q'\,tt\}\; t_2\; \{Q\}}{\{H\}\; (t_1\,;\,t_2)\; \{Q\}}$$

The characteristic formula $[\![t_1\,;\,t_2]\!]$ associated with the term "$t_1\,;\,t_2$" should be a predicate such that the proposition $[\![t_1\,;\,t_2]\!]\,H\,Q$ is equivalent to the triple $\{H\}\;(t_1\,;\,t_2)\;\{Q\}$. Thus, $[\![t_1\,;\,t_2]\!]$ should be an abstraction that takes H and Q as argument. The body of the definition should match the premises of the reasoning rules shown above: there should exist a Q' such that t_1 admits H and Q' as specification and such that t_2 admits $Q'\,tt$ and Q as specification. We therefore consider the following definition.

$$[\![t_1\,;\,t_2]\!] \;\equiv\; \lambda H.\,\lambda Q.\;\; \exists Q'.\;\; [\![t_1]\!]\,H\,Q' \;\;\wedge\;\; [\![t_2]\!]\,(Q'\,tt)\;Q$$

In the intention, the characteristic formula closely resembles the corresponding Hoare-logic rule. However, the characteristic formulae correspond to the recursive definition of a predicate, whereas Hoare-logic rules correspond to inductive definitions. Moreover, in the characteristic formula, the intermediate post-condition Q' is explicitly introduced with an existential quantifier, whereas this quantification is implicit in the Hoare-logic derivation rule. The existential quantification of unknown specifications, which is made possible by the strength of higher-order logic, plays a central role. In particular, this existential quantification of specifications contrasts with traditional program verification approaches where intermediate specifications, including loop invariants, need to be included in the source code.

We introduce a notation system for pretty-printing characteristic formulae, whose purpose is to make proof obligations easily readable and closely related to the source code. For sequences, we define the piece of notation shown below. Bold keywords correspond to notation for logical formulae, whereas plain keywords correspond to constructors from the programming language syntax.

$$(\mathcal{F}_1\,;\mathcal{F}_2) \;\equiv\; \lambda H.\,\lambda Q.\;\; \exists Q'.\;\, \mathcal{F}_1\,H\,Q' \;\;\wedge\;\; \mathcal{F}_2\,(Q'\,tt)\;Q$$

The definition of the characteristic formula of a let-binding can now be reformulated as follows.

$$[\![t_1\,;\,t_2]\!] \;\equiv\; ([\![t_1]\!]\,;[\![t_2]\!])$$

The generation of characteristic formulae, which is a translation from program syntax to higher-order logic, therefore boils down to a re-interpretation of the programming language keywords in terms of logical predicates.

The construction pattern described for sequences generalizes to other language constructs, including constructs with binders. For example, consider a let-binding of the form "$\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2$". The reasoning rule for let-bindings can be stated as follows. (Note that x denotes a logical variable in the premise although it denotes a program variable in the conclusion.)

$$\frac{\{H\}\;t_1\;\{Q'\} \qquad \forall x.\;\{Q'\,x\}\;t_2\;\{Q\}}{\{H\}\;(\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2)\;\{Q\}}$$

The characteristic formula for let-bindings is as follows.

$$[\![\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2]\!] \;\equiv\; \lambda H.\,\lambda Q.\;\; \exists Q'.\;\, [\![t_1]\!]\,H\,Q' \;\;\wedge\;\forall x.\;\, [\![t_2]\!]\,(Q'\,x)\;Q$$

Here again, the construction of the formula is entirely compositional and we are able to reformulate the definition of the characteristic formula using an appropriate piece of notation.

$$(\textbf{let}\;x = \mathcal{F}_1\;\textbf{in}\;\mathcal{F}_2) \;\equiv\; \lambda H.\,\lambda Q.\;\; \exists Q'.\;\, \mathcal{F}_1\,H\,Q' \;\;\wedge\;\forall x.\;\, \mathcal{F}_2\,(Q'\,x)\;Q$$

$$[\![\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2]\!] \;\equiv\; (\textbf{let}\;x = [\![t_1]\!]\;\textbf{in}\;[\![t_2]\!])$$

Similarly, characteristic formulae and notation can be defined for all the other constructions of the programming language. It follows that characteristic formulae may be pretty-printed exactly like the source code they describe. During the verification of a term t, the proof-obligation takes the form $[\![t]\!]\,H\,Q$ and thus appears to the user as a piece of source code followed with its pre-condition and its post-condition, that is, it reads as "$t\,H\,Q$". Note that this convenient display applies not only to a top-level program definition t but also to all of the subterms of t involved during the verification of t.

CFML provides a set of tactics for making progress in the analysis of a characteristic formula. For example, the tactic `xseq` applies to a goal of the form "$(\mathcal{F}_1\,;\mathcal{F}_2)\,H\,Q$". It introduces a fresh unification variable, call it Q', and produces two subgoals: $\mathcal{F}_1\,H\,Q'$ and $\mathcal{F}_2\,(Q'\,tt)\,Q$. The intermediate specification Q' introduced here typically gets instantiated through unification when solving the first subgoal. The pre-condition for $\mathcal{F}_2$ is thus known when starting to reason about the second subgoal. The instantiation of Q' may also be provided by the user explicitly, as argument of the tactic `xseq`. More generally, CFML provides one such "x-tactic" for each language construction. As a result, the user is able to verify a program using characteristic formulae even without any knowledge about the construction of characteristic formulae.

2.3 Integration of the frame rule

Local reasoning [40] refers to the ability to verify a piece of code by reasoning only about the memory cells that are involved in the execution of this code. With local reasoning, all the memory cells that are not explicitly mentioned are implicitly assumed to remain unchanged. The concept of local reasoning is very elegantly captured by the "frame rule", which originates in Separation Logic [43].

The frame rule states that if a program expression transforms a heap described by a predicate H_1 into heap described by a predicate H_1', then, for any heap predicate H_2, the same program expression also transforms a heap of the form $H_1 * H_2$ into a state of the form $H_1' * H_2$. The star symbol, called separating conjunction, captures a disjoint union of two pieces of heap. The frame rule can be formulated in terms of Hoare triples as shown next.

$$\frac{\{H_1\}\ t\ \{Q_1\}}{\{H_1 * H_2\}\ t\ \{Q_1 \star H_2\}}$$

Above, the symbol ($\star$) is like ($*$) except that it extends a post-condition with a piece of heap. Technically, $Q_1 \star H_2$ is defined as "$\lambda x.\,(Q_1\,x) * H_2$", where the variable x denotes the output value and $Q_1\,x$ describes the output heap.

The frame rule is not syntax-directed, meaning that one cannot guess from the shape of the term t when the frame rule needs to be applied. Yet, our goal is to generate characteristic formulae in a systematic manner from the syntax of the source code. So, we introduce a predicate transformer, called local, which corresponds to an application of the frame rule, and we insert this predicate at every node of a characteristic formula. This predicate is defined in such a way that, to prove the proposition "$\mathsf{local}\,[\![t]\!]\,H\,Q$", it suffices to find a decomposition of H of the form $H_1 * H_2$, a decomposition of Q of the form $Q_1 \star H_2$, and to prove $[\![t]\!]\,H_1\,Q_1$. In first approximation, the predicate local is defined as follows.

$$\begin{array}{rcl}\mathsf{local}\,\mathcal{F} & \equiv & \lambda H.\,\lambda Q.\ \exists H_1.\,\exists H_2.\,\exists Q_1. \\ & & \quad H = H_1 * H_2 \ \wedge\ \mathcal{F}\,H_1\,Q_1 \ \wedge\ Q = Q_1 \star H_2\end{array}$$

This predicate is inserted at the head of every characteristic formulae, and our pieces of notation are updated accordingly. For example, the notation for let-bindings is updated as follows.

$$(\mathbf{let}\ x = \mathcal{F}_1\ \mathbf{in}\ \mathcal{F}_2) \ \equiv\ \mathsf{local}\,(\lambda H.\,\lambda Q.\ \exists Q'.\ \mathcal{F}_1\,H\,Q' \wedge \forall x.\ \mathcal{F}_2\,(Q'\,x)\,Q)$$

The introduction of the predicate local throughout characteristic formulae allows us to apply the frame rule at any time during program verification. If there is no need to apply the frame rule, then the local predicate may be simply ignored. Indeed, given a formula $\mathcal{F}$, the proposition "$\mathcal{F}\,H\,Q$" is always a sufficient condition for proving "$\mathsf{local}\,\mathcal{F}\,H\,Q$". It suffices to instantiate H_2 as the empty heap predicate. We will later generalize the local predicate so as to also handle applications of the rule of consequence, which is used to strengthen pre-conditions and weaken post-conditions, and to handle the garbage collection rule, which allows to discard memory cells from assertions on heaps.

2.4 Translation of types

Purely-functional values may be directly reflected in the logic. For example, functional lists of Caml perfectly match the the lists of Coq, so we can use the latter to reason about the former. However, particular care is required for specifying and reasoning about program functions. Indeed, program functions cannot be directly represented as logical functions, because of a profound mismatch between the two: program functions may be partial, whereas logical functions must always be total. To work around this mismatch, we introduce a new data type, called Func, in order to represent program functions in the logic. We present the type Func as an abstract data type to the user of characteristic formulae. In the proof of soundness, we interpret a value of type Func as the syntax of the source code of a function.

To represent pointers in the logic, we simply view them as memory locations. We use an abstract data type called Loc to represent these locations. In the soundness proof, loc is implemented using natural numbers. Note that, contrary to the type $\mathsf{ref}\,\tau$ of Caml, the type Loc does not carry any information about the type of the value stored in memory. Indeed, the type and the content of a memory cell is instead described explicitly using heap predicates, so there is no need to constrain the type Loc.

The translation of Caml types into Coq types is formalized through an operator, written $\langle\cdot\rangle$. In particular, this operator maps all arrow types to the type Func and maps all reference types to the type Loc. For simplicity, we assume that the compiler implements integers using an arbitrary-precision representation, so we simply map Caml values of type int to Coq values of type $\mathbb{Z}$. Note that it would also be possible to map the type int to the Coq type $\mathsf{int64}$. The definition of the operator $\langle\cdot\rangle$ is summarized as follows.

$$\begin{array}{lcl}
\langle \mathsf{int} \rangle & \equiv & \mathbb{Z} \\
\langle \tau_1 \times \tau_2 \rangle & \equiv & \langle \tau_1 \rangle \times \langle \tau_2 \rangle \\
\langle \tau_1 + \tau_2 \rangle & \equiv & \langle \tau_1 \rangle + \langle \tau_2 \rangle \\
\langle \tau_1 \rightarrow \tau_2 \rangle & \equiv & \mathsf{Func} \\
\langle \mathsf{ref}\,\tau \rangle & \equiv & \mathsf{Loc}
\end{array}$$

The translation from Caml types to Coq types is in fact conducted in two steps. A well-typed ML program gets first translated into a well-typed *weak-ML* program, and this weak-ML program is then fed to the characteristic formula generator. Weak-ML corresponds to a relaxed version of ML that does not keep track of the type of pointers nor of the type of functions. Moreover, weak-ML does not impose any constraint on the typing of applications nor on the typing of dereferencing.

Since weak-ML imposes strictly fewer constraints than ML, any program well-typed in ML is also well-typed in weak-ML. Weak-ML nevertheless enforces strong enough invariants to justify the soundness of characteristic formulae. So, even though memory safety is not guaranteed by weak-ML, it is guaranteed by the proofs of correctness established using a characteristic formula generated from a well-typed weak-ML program.

It would be possible to generate characteristic formulae directly from ML programs. Yet, the use of weak-ML as an intermediate type system serves three important purposes. First, weak-ML helps simplifying the definition of the characteristic formula generation algorithm. Second, it enables the verification of programs that are well-typed in weak-ML but not in ML, such as programs exploiting System F functions, null pointers, or strong updates (i.e., type-varying updates of a reference cell). Third, weak-ML plays a crucial role in proving the soundness and completeness of characteristic formulae. This latter aspect of weak-ML is not discussed in this paper. It is, however, described in the author's PhD dissertation [8].

2.5 Reasoning about functions

To specify the behavior of functions, we rely on a predicate, called App, which we also present to the user as an abstract predicate. Intuitively, the proposition "$\mathsf{App}\, f\, v\, H\, Q$" asserts that the application of the function f to v in a heap satisfying H terminates and returns a value v' in a heap satisfying $Q\, v'$. The predicates H and Q correspond to the pre- and post-conditions of the application of the function f to the argument v. It follows that the characteristic formula for an application of a function f to a value v is simply built as the partial application of App to f and v.

$$[\![f\, v]\!] \quad \equiv \quad \mathsf{App}\, f\, v$$

The function f is viewed in the logic as a value of type Func. If f takes as argument a value v described in Coq at type A and returns a value described in Coq at type B, then the pre-condition H has type Hprop, a shorthand for $\mathsf{Heap} \to \mathsf{Prop}$, and the post-condition Q has type $B \to \mathsf{Hprop}$. So, the predicate App is typed as follows.

$$\mathsf{App} \;\; : \;\; \forall A\, B.\, \mathsf{Func} \to A \to \mathsf{Hprop} \to (B \to \mathsf{Hprop}) \to \mathsf{Prop}$$

For example, recall the triple describing the behavior of the increment function: $\{r \mapsto n\}\ (\mathsf{incr}\, r)\ \{\lambda\_.\ r \mapsto n+1\}$. This specification is expressed with the predicate App as follows.

$$\forall r.\ \forall n.\ \mathsf{App}\, \mathsf{incr}\, r\, (r \mapsto n)\, (\lambda\_.\ r \mapsto n+1)$$

As we have just seen, a statement of the form "$\mathsf{App}\, f\, v\, H\, Q$" describes the behavior of an application and can be used to state specifications. It remains to explain where assumptions of the form "$\mathsf{App}\, f\, v\, H\, Q$" can be obtained from. Such assumptions are provided by characteristic formulae associated with function definitions. Consider a function f defined as the abstraction "$\lambda x.\, t$". CFML represents this function in Coq by introducing an abstract constant (i.e., a Coq axiom) named f of type Func. Given a particular argument v, we expect to be able to derive an instance of "$\mathsf{App}\, f\, v\, H\, Q$" simply by proving that the body t, in which x is instantiated with v, admits the pre-condition H and the post-condition Q. To that end, CFML provides a second abstract constant to describe the semantics of the function: "$\forall x\, H\, Q.\ [\![t]\!]\, H\, Q \Rightarrow \mathsf{App}\, f\, x\, H\, Q$". Instantiating the variable x with a value v in this constant automatically performs the appropriate substitution in the characteristic formula $[\![t]\!]$ in which x may occur as a free variable. Note that the soundness theorem proved for characteristic formulae ensures that adding these two axioms does not introduce any logical inconsistency.

For example, assume that f is defined as a function that expects a reference and increments its content twice: "$\lambda r.\, (\mathsf{incr}\, r\, ;\, \mathsf{incr}\, r)$". This function may be specified through the theorem "$\forall rn.\ \mathsf{App}\, f\, r\, (r \hookrightarrow n)\, (\lambda\_.\ r \hookrightarrow n+2)$". To prove this theorem, we apply the axiom generated by CFML. The resulting proof obligation is "$(\textbf{app}\, \mathsf{incr}\, r\, \mathbf{;}\ \textbf{app}\, \mathsf{incr}\, r)\, (r \hookrightarrow n)\, (\lambda\_.\ r \hookrightarrow n+2)$" where "**app**" and "**;**" correspond to the pieces of notation defined for the characteristic formulae of applications and of sequences, respectively. This proof obligation can be discharged with help of the tactic `xseq`, for reasoning about the sequence, and of the tactic `xapp`, for reasoning about the two applications. In fact, for such a simple function, one may establish correctness through a simple invocation of a tactic called `xgo`, which repeatedly applies the appropriate x-tactic until some information is required from the user.

When a function is not a top-level definition but a local definition, we generate its characteristic formula as follows.

$$\begin{array}{rcl} [\![\mathsf{let\ rec}\, f = \lambda x.\, t\, \mathsf{in}\, t']\!] & \equiv & \lambda H.\, \lambda Q.\ \forall f.\ \mathcal{H} \Rightarrow [\![t']\!]\, H\, Q \\ \text{where } \mathcal{H} & \equiv & (\forall x\, H'\, Q'.\ [\![t]\!]\, H'\, Q' \Rightarrow \mathsf{App}\, f\, x\, H'\, Q') \end{array}$$

Two observations are worth making about the treatment of functions. First, characteristic formulae do not involve any specific treatment of recursivity. Indeed, to prove that a recursive function satisfies a given specification, it suffices to conduct a proof that the function satisfies that specification by induction. The induction may be conducted on a measure or on a well-founded relation, using the induction facility from the interactive theorem prover being used. So, characteristic formulae for recursive functions do not need to include any induction hypothesis. A similar observation was also made by Honda *et al* in their work on program logics [21].

The second observation concerns first-class functions. As explained through this section, a function f is specified with a statement of the form "$\mathsf{App}\, f\, v\, H\, Q$". Because this statement is a proposition like any other (it has type Prop), it may appear inside the pre-condition or the post-condition of any another function (thanks to the impredicativity of Prop in Coq). Moreover, this statement may appear in the specification of the content of a memory cell. The predicate App therefore supports reasoning about higher-order functions (functions taking functions as arguments) and higher-order stores (memory stores containing functions).

3 Characteristic formula generation

In this section, we explain in more details how characteristic formulae are constructed. We start with the translation from ML types into weak-ML types and into Coq types. We then describe the source language, the representation in Coq of program values and heaps, and the construction of characteristic formulae.

3.1 Translation of types

In what follows, we formalize the translation from ML types to weak-ML types, then the translation from weak-ML types to Coq types. We let τ denote an ML type, σ denote an ML type scheme, A denote a type variable, and C denote a type constructor associated with an algebraic data type. We use the overbar notation to denote a list of items. The grammar of ML types is as follows.

$$\begin{array}{lcl}\tau & := & A \;\mid\; \mathsf{int} \;\mid\; C\,\overline{\tau} \;\mid\; \tau \to \tau \;\mid\; \mathsf{ref}\,\tau \;\mid\; \mu A.\tau \\ \sigma & := & \forall \overline{A}.\tau\end{array}$$

Note that sum types, product types, the boolean type and the unit type can be defined as algebraic data types.

Intuitively, weak-ML types are obtained from ML types by mapping all arrow types to a constant type called func and by mapping all reference types to a constant type called loc. We let T denote a weak-ML type and S denote a weak-ML type scheme. The grammar of weak-ML types is as follows.

$$\begin{array}{lcl}T & := & A \;\mid\; \mathsf{int} \;\mid\; C\,\overline{T} \;\mid\; \mathsf{func} \;\mid\; \mathsf{loc} \\ S & := & \forall \overline{A}.T\end{array}$$

The formalization of the translation of an ML type τ into its corresponding weak-ML type, written $\langle \tau \rangle$, appears in Figure 1. The treatment of polymorphism and of recursive types is explained next. When translating a type scheme, the list of quantified variables might shrink. For example, the ML type scheme "$\forall AB.\ A + (B \to B)$" is mapped in weak-ML to "$\forall A.\ A + \mathsf{func}$", which no longer involves the type variable B. Weak-ML includes algebraic data types, but does not support general equi-recursive types. For example, the recursive ML type "$\mu A.(A \times \mathsf{int})$" does not have any counterpart in weak-ML. Nevertheless, many useful recursive ML types can be translated into weak-ML, because the recursion involved might vanish when erasing arrow types. For example, the recursive ML type "$\mu A.(A \to B)$" gets mapped to the weak-ML type func. More generally,

$$\begin{array}{lcl}
\langle A \rangle & \equiv & A \\
\langle \mathsf{int} \rangle & \equiv & \mathsf{int} \\
\langle C\,\overline{\tau} \rangle & \equiv & C\,\langle \overline{\tau} \rangle \\
\langle \tau_1 \to \tau_2 \rangle & \equiv & \mathsf{func} \\
\langle \mathsf{ref}\,\tau \rangle & \equiv & \mathsf{loc} \\
\langle \forall \overline{A}.\, \tau \rangle & \equiv & \forall \overline{B}.\, \langle \tau \rangle \quad \text{where } \overline{B} = \overline{A} \cap \mathsf{fv}(\langle \tau \rangle)
\end{array}$$

For equi-recursive types that are not algebraic data types:

$$\langle \mu A.\tau \rangle \equiv \left| \begin{array}{ll} \langle \tau \rangle & \text{if } A \notin \langle \tau \rangle \\ \text{program rejected otherwise} & \end{array} \right.$$

Fig. 1. Translation from ML types to weak-ML types.

our approach supports reasoning about any function with a recursive type. In fact, we could even support System F functions if the source language was not restricted to ML type schemes.

When building the characteristic formula of a weak-ML program, weak-ML types get translated into Coq types. This translation is almost the identity, because every type constructor from weak-ML is directly mapped to the corresponding Coq type constructor. Algebraic type definitions are translated into corresponding Coq inductive definitions. The translation of a weak-ML type T into its corresponding Coq type, written $[\![T]\!]$, is defined as follows.

$$\begin{array}{lcl|lcl}
[\![\mathsf{int}]\!] & \equiv & \mathbb{Z} & [\![A]\!] & \equiv & A \\
[\![\mathsf{loc}]\!] & \equiv & \mathsf{Loc} & [\![C\,\overline{T}]\!] & \equiv & C\,[\![\overline{T}]\!] \\
[\![\mathsf{func}]\!] & \equiv & \mathsf{Func} & [\![\forall \overline{A}.T]\!] & \equiv & \forall \overline{A}.\, [\![T]\!]
\end{array}$$

Above, the type variables $\overline{A}$ are assigned the kind Type in Coq —we will come back to this aspect in §4.3. Note that the positivity requirement associated with Coq inductive definitions is not a problem here: since there is no arrow type in weak-ML, the translation from weak-ML types to Coq types can never produce a negative occurrence of an inductive type in its own definition.

3.2 Typed source language

Before we can generate characteristic formulae, we first need to put programs in *administrative normal form*. To that end, we pull out all effectful subexpressions and name their results using let-bindings. We do the same for definitions of local functions. This process, similar to A-normalization [15], preserves the semantics and greatly simplifies formal reasoning about programs. Similar transformations have appeared in previous work on program verification (e.g., [21, 42]). In this paper, we omit a formal description of the normalization process and only show the grammar of terms in normal form.

The characteristic formula generator expects a program in administrative normal form. It moreover expects this program to be well-typed in weak-ML, with every subterm being annotated with its type. We show below the syntax of typed programs in normal forms. We let $\hat{v}$ range over typed values, $\hat{t}$ range over typed term, $\hat{b}$ range over lists of pattern matching clauses and $\hat{p}$ range over typed patterns.

$$\begin{array}{lcl}
\hat{v} & := & n \;\mid\; x\,\overline{T} \;\mid\; D\,\overline{T}(\hat{v},\ldots,\hat{v}) \;\mid\; \mathsf{ref} \;\mid\; \mathsf{get} \;\mid\; \mathsf{set} \;\mid\; \mathsf{cmp} \;\mid\; \mathsf{null} \\
\hat{t} & := & \hat{v} \;\mid\; (\hat{v}\,\hat{v}) \;\mid\; \mathsf{if}\,\hat{v}\,\mathsf{then}\,\hat{t}\,\mathsf{else}\,\hat{t} \;\mid\; \mathsf{let}\,x = \hat{t}\,\mathsf{in}\,\hat{t} \;\mid\; \mathsf{let}\,x = \Lambda\overline{A}.\,\hat{v}\,\mathsf{in}\,\hat{t} \;\mid \\
 & & \hat{t}\,;\,\hat{t} \;\mid\; \mathsf{let\,rec}\,f = \Lambda\overline{A}.\lambda x.\hat{t}\,\mathsf{in}\,\hat{t} \;\mid\; \mathsf{assert}\;\hat{t} \;\mid\; \mathsf{match}\,\hat{v}\,\mathsf{with}\,\hat{b} \\
\hat{b} & := & \emptyset \;\mid\; (\hat{p} \mapsto \hat{t} \,|\, \hat{b}) \\
\hat{p} & := & x \;\mid\; n \;\mid\; D\,\overline{T}(\hat{p},\ldots,\hat{p})
\end{array}$$

Note that locations and function closures do not exist in source programs, so they are not included in the grammar above. The letter n denotes an integer. The functions ref, get and set are used to allocate, read and write reference cells, respectively. The function cmp enables comparison of two memory locations. The null pointer, written null, is a particular location that never gets allocated. A polymorphic function definition takes the form "$\mathsf{let\,rec}\,f = \Lambda\overline{A}.\lambda x.\hat{t}_1\;\mathsf{in}\,\hat{t}_2$", where $\overline{A}$ denotes the list of generalized type variables. A polymorphic let-binding takes the form "$\mathsf{let}\,x = \Lambda\overline{A}.\,\hat{v}\,\mathsf{in}\,\hat{t}$". Due to the value restriction, the general form "$\mathsf{let}\,x = \Lambda\overline{A}.\,\hat{t}_1\,\mathsf{in}\,\hat{t}_2$" is not allowed. Observe that the syntax of typed programs explicitly keeps track of type applications, which take place either on a polymorphic variable x, written $x\,\overline{T}$, or on a polymorphic data constructor D, written $D\,\overline{T}$. For-loops and while-loops are discussed further on (§3.10).

3.3 Reflection of values in the logic

Constructing characteristic formulae requires a translation of every Caml value that appears in the program source code into its corresponding Coq value. This translation, called *decoding*, and written $\lceil\hat{v}\rceil$, transforms a weak-ML value $\hat{v}$ of type T into the corresponding Coq value, which has type $[\![T]\!]$ (recall §3.1). The definition of $\lceil\hat{v}\rceil$ is shown below and explained next. Values on the left-hand side denote well-typed weak-ML values, and values on the right-hand side denote (well-typed) Coq values.

$$\begin{array}{lcl}
\lceil n\rceil & \equiv & n \\
\lceil x\,\overline{T}\rceil & \equiv & x\,[\![\overline{T}]\!] \\
\lceil D\,\overline{T}(\hat{v}_1,\ldots,\hat{v}_2)\rceil & \equiv & D\,[\![\overline{T}]\!]\,(\lceil\hat{v}_1\rceil,\ldots,\lceil\hat{v}_2\rceil) \\
\lceil\Lambda\overline{A}.\,\hat{v}\rceil & \equiv & \lambda\overline{A}.\,\lceil\hat{v}\rceil
\end{array}$$

A program integer n is mapped to the corresponding Coq integer. If x is a non-polymorphic variable, then it is simply mapped to itself. However, if x is a polymorphic variable applied to some types $\overline{T}$, then this occurrence is translated as the iterated application of x to the translation of each of the types from the list $\overline{T}$. A program data constructor D is mapped to the corresponding Coq inductive constructor. Polymorphic data constructors, like polymorphic variables, need to be applied to the appropriate lists of types. The primitive functions for manipulating references (e.g., get) are mapped to corresponding abstract Coq values of type Func.

The decoding of a polymorphic value $\Lambda\overline{A}.\hat{v}$ is a Coq function that expects the types $\overline{A}$ and returns the decoding of the value $\hat{v}$. For example, the polymorphic pair $(\mathsf{nil}, \mathsf{nil})$ has type "$\forall A.\, \forall B.\, \mathsf{list}\, A \times \mathsf{list} B$". The Coq translation of this value is "`fun (A B : Type) => (@nil A, @nil B)`", where the prefix `@` indicates that type arguments are given explicitly.

3.4 Heap predicates

The semantics of a source program involves a memory store, which is a finite map from locations to program values. We represent the memory store in Coq as a *heap* data structure. In what follows, we describe the formalization in Coq of heaps and of heap predicates in the style of Separation Logic.

The type Heap is defined in Coq as the type of finite maps from locations to dependent pairs, where a dependent pair is a pair made of a Coq type $\mathcal{T}$ and of a Coq value V of type $\mathcal{T}$. With this definition and the notion of exotic values defined further on (§4.3), we are able to establish a bijection between the set of well-typed memory stores and the set Coq values of type Heap.

We define operations on heaps in terms of operations on maps. The empty heap, written $\varnothing$, is defined as the empty map. A singleton heap, written $l \rightarrow_{\mathcal{T}} V$, is defined as a singleton map that binds a location l to a dependent pair made of a type $\mathcal{T}$ and a value V of type $\mathcal{T}$. We say that two heaps are disjoint, written $h_1 \perp h_2$, when their underlying maps have disjoint domains. We define the union of two heaps, written $h_1 + h_2$, as the union of the two underlying finite maps. Note that we are only concerned with disjoint unions here, so we do not need to specify the behavior of the union operator on two maps with overlapping domains.

Using these basic operations on heaps, we define predicates for specifying heaps in the style of Separation Logic, as is done for example in Ynot [11]. Heap predicates are predicates over values of type Heap, so they have the type $\mathsf{Heap} \rightarrow \mathsf{Prop}$, which we abbreviate as Hprop. A singleton heap that binds a non-null location l to a value V of type $\mathcal{T}$ is characterized by the predicate $l \hookrightarrow_{\mathcal{T}} V$. This predicate is defined as $\lambda h.\ l \neq \mathsf{null} \wedge h = (l \rightarrow_{\mathcal{T}} V)$. The heap predicate $H_1 * H_2$ holds of a disjoint union of a heap satisfying H_1 and of a heap satisfying H_2. It is formally defined as follows.

$$H_1 * H_2 \quad \equiv \quad \lambda h.\ \exists h_1 h_2.\ h_1 \perp h_2 \ \wedge \ h = h_1 + h_2 \ \wedge \ H_1\, h_1 \ \wedge \ H_2\, h_2$$

We lift propositions to the world of heap predicates by defining the predicate $[\mathcal{P}]$ to describe an empty heap and carry the information that the proposition $\mathcal{P}$ is true. Formally, we define $[\mathcal{P}]$ as $\lambda h.\ \mathcal{P} \wedge (h = \varnothing)$. We abbreviate $[\mathsf{True}]$ as $[\,]$. We also lift existential quantifiers: $\exists x.\, H$ holds of a heap h if there exists a value x such that H holds of that heap. The formal definition of existential quantifiers properly handles binders: $\exists x.\, H$ is in fact a notation for $\mathsf{hexists}\,(\lambda x.\, H)$, where $\mathsf{hexists}$ is defined as shown below (where the first argument is always left implicit).

$$\mathsf{hexists}\,(A : \mathsf{Type})\,(J : A \to \mathsf{Hprop}) \;\equiv\; \lambda(h : \mathsf{Heap}).\, \exists(x : A).\, J\,x\,h$$

In this work, we ignore the disjunction construct $(H_1 \vee H_2)$. Instead, to reason about the content of a heap by case analysis, we rely on heap predicates of the form "$\mathsf{if}\,P\,\mathsf{then}\,H_1\,\mathsf{else}\,H_2$", which are defined using the built-in conditional construct from classical logic. We also do not make use of the non-separating conjunction $(H_1 \wedge H_2)$, and do not use the conjunction rule, which can be found other formalizations of Separation Logic. From a practical perspective, we never felt the need for the conjunction rule and we would find it very hard to devise tactics to manipulate it. From a theoretical perspective, the conjunction rule is not needed for characteristic formulae to achieve completeness. Finally, if we wanted to include a conjunction rule, we would need to adapt it in a nontrivial way, otherwise it would not be compatible with garbage collection. For example, using the garbage collection rule through the predicate local defined further on (§3.6), we can prove $[\![\mathsf{ref}\,3]\!]\;[\,]\;(\lambda r.\, r \hookrightarrow 3)$ and prove $[\![\mathsf{ref}\,3]\!]\;[\,]\;(\lambda r.\, [\,])$, however there does not exist any heap that satisfies both $r \hookrightarrow 3$ and $[\,]$ at the same time. For all the aforementioned reasons, we are not interested in the non-separating conjunction $(H_1 \wedge H_2)$.

Two heap predicates that characterize exactly the same set of heaps can be proved to be *equal* thanks to the axiom of predicate extensionality, which we assume in our work. Predicate extensionality asserts that the implication $(\forall x.\, P\,x \Leftrightarrow Q\,x) \Rightarrow (P = Q)$ holds for any predicates P and Q.[2] Using predicate extensionality, we can show that heap predicates form a commutative monoid, and that existential quantifiers commute with the star operation under appropriate freshness conditions. These properties are formalized as follows.

$$\begin{array}{ll}
\text{NEUTRAL:} & H * [\,] = H \\
\text{COMMUTATIVITY:} & H_1 * H_2 = H_2 * H_1 \\
\text{ASSOCIATIVITY:} & (H_1 * H_2) * H_3 = H_1 * (H_2 * H_3) \\
\text{SCOPE EXTRUSION:} & (\exists x.\, H_1) * H_2 = \exists x.\, (H_1 * H_2) \quad \text{when } x \notin \mathsf{fv}(H_2)
\end{array}$$

3.5 Heap entailment relation

To define characteristic formulae, we need to use a *heap entailment relation.* This relation, written $H_1 \rhd H_2$, asserts that any heap satisfying H_1 also satisfies H_2. It is formally defined as $\forall h.\, H_1\,h \Rightarrow H_2\,h$. By extension, we define the entailment relation for post-conditions, written $Q_1 \blacktriangleright Q_2$, and defined as $\forall x.\, Q_1\,x \rhd Q_2\,x$.

[2] Predicate extensionality, instead of being taken as an axiom, may also be derived from two lower-level axioms: (1) functional extensionality, which asserts that, for any functions f and g, we have $(\forall x.\, f\,x = g\,x) \Rightarrow (f = g)$, and (2) propositional extensionality, which asserts that, for any propositions P and Q, we have $(P \Leftrightarrow Q) \Rightarrow (P = Q)$.

Heap entailment yields a partial order on heap predicates. Moreover, heap entailment is regular with respect to the star operator.

REFLEXIVITY:	$H \rhd H$
TRANSITIVITY:	$H_1 \rhd H_2 \;\wedge\; H_2 \rhd H_3 \;\Rightarrow\; H_1 \rhd H_3$
ANTISYMMETRY:	$H_1 \rhd H_2 \;\wedge\; H_2 \rhd H_1 \;\Rightarrow\; H_1 = H_2$
REGULARITY:	$H_1 \rhd H_2 \;\wedge\; H_1' \rhd H_2' \;\Rightarrow\; H_1 * H_1' \rhd H_2 * H_2'$

The regularity property is typically exploited to cancel out a same heap predicate from both sides of a heap entailment relation. For example, using REGULARITY and REFLEXIVITY we can easily show that in order to prove $H_1 * H_2 * H_3 \rhd H_4 * H_2 * H_5$, we may cancel out H_2 from both sides: it is sufficient to prove $H_1 * H_3 \rhd H_4 * H_5$.

Furthermore, we need lemmas to extract and instantiate existential quantifiers and pure facts from both sides of a heap entailment relating a source and a target heap. These lemmas are shown below (omitting the classic freshness side conditions) and explained next.

EXISTS-LEFT:	$(\forall x.\ (H_1 \rhd H_2)) \;\Rightarrow\; (\exists x. H_1) \rhd H_2$
PROP-LEFT:	$(\mathcal{P} \;\Rightarrow\; (H_1 \rhd H_2)) \;\Rightarrow\; ([\mathcal{P}] * H_1) \rhd H_2$
EXISTS-RIGHT:	$H_1 \rhd ([x \to v]\, H_2) \;\Rightarrow\; H_1 \rhd (\exists x. H_2)$
PROP-RIGHT:	$(H_1 \rhd H_2) \;\wedge\; \mathcal{P} \;\Rightarrow\; H_1 \rhd (H_2 * [\mathcal{P}])$

The lemma EXTRACT-EXISTS-LEFT allows to extract an existential quantifier packed in the source heap; this existential quantifier, by contra-variance becomes a universal quantifier outside of the judgment; EXTRACT-PROP-LEFT allows to extract an assumption about the source heap and to add this assumption to the current proof context; EXTRACT-EXISTS-RIGHT allows to instantiate an existential quantifier in front of the target heap with a particular value; EXTRACT-PROP-RIGHT allows to provide a proof of a pure fact attached to the target heap.

All these definitions are formalized in Coq and are usually exploited automatically by tactics. For example, to prove $\exists k. (r \hookrightarrow 4k) \rhd \exists n. (r \hookrightarrow n) * [\mathsf{even}\, n]$, our tactic first applies the rule EXTRACT-EXISTS-LEFT to consider an arbitrary k and introduce it in the proof context. It applies the rule EXTRACT-EXISTS-RIGHT to instantiate n with a fresh Coq unification variable, call it N. It applies the rule EXTRACT-PROP-RIGHT. It then solves the goal $(r \hookrightarrow 4k) \rhd (r \hookrightarrow N)$ by REFLEXIVITY, instantiating N with $4k$ in the process. It then leaves the goal $\mathsf{even}\,(4k)$ for the user to discharge (possibly using another automated tactic).

Note that the reasoning about heap involved for manipulating characteristic formulae never requires manipulating values of type Heap directly: the reasoning exclusively takes place at the level of heap predicates, in terms of the heap entailment relation.

Remark: the Separation Logic that we use here is not intuitionist: in general, the entailment $H_1 * H_2 \rhd H_1$ is false. In our work, garbage collection is taken care of at the level of characteristic formulae with an explicit quantification over pieces of heap to be discarded.

3.6 Local formulae

In the introduction, we suggested how to define the predicate transformer "local" to account for applications of the frame rule. We now present a more general definition of this predicate, which also accounts for the rule of consequence and for the rule of garbage collection, and that supports the extraction of propositions and existentially-quantified variables from pre-conditions.

The definition of $\mathsf{local}\,\mathcal{F}\,H\,Q$, where $\mathcal{F}$ is a formula of the type $\mathsf{Hprop} \to (A \to \mathsf{Hprop}) \to \mathsf{Prop}$, consists of generalizing the definition given in the introduction in order to add the possibility for strengthening the pre-condition, weakening the post-condition, and performing garbage collection. Let H and Q describe the initial and the final heap, H_1 and Q_1 describe the portions of H and Q with which the formula $\mathcal{F}$ is concerned, let H_2 correspond to the part of the heap that is being framed out, and H_3 correspond to the part of the heap that gets garbage-collected at the logical level. A first attempt at the definition of local is as follows.

$$\mathsf{local}'\,\mathcal{F}\,H\,Q \;\;\equiv\;\; \exists H_1 H_2 H_3 Q_1.\; H \rhd H_1 * H_2 \;\;\wedge\;\; \mathcal{F}\,H_1\,Q_1 \;\;\wedge\;\; Q_1 \star H_2 \blacktriangleright Q \star H_3$$

Yet, this definition is not strong enough to support the extraction of propositions and of existentially-quantified variables from pre-conditions (lemmas EXTRACT-PROP and EXTRACT-EXISTS stated further on). To support them, we need to explicitly quantify over the input heap. The appropriate definition is as follows.

$$\begin{array}{rcl}\mathsf{local}\,\mathcal{F} & \equiv & \lambda H\,Q.\; \forall h.\; H\,h \Rightarrow \exists H_1 H_2 H_3 Q_1. \\ & & \qquad (H_1 * H_2)\,h \;\;\wedge\;\; \mathcal{F}\,H_1\,Q_1 \;\;\wedge\;\; Q_1 \star H_2 \blacktriangleright Q \star H_3\end{array}$$

Note that the definition of the predicate local shows some similarities with the definition of the "STsep" monad from Hoare Type Theory [36], in the sense that both aim at baking the Separation Logic frame condition into a system originally defined in terms of heaps describing the whole memory.

We can prove that the predicate local may safely be discarded during reasoning, in the sense that "$\mathcal{F}\,H\,Q$" is a sufficient condition for proving "$\mathsf{local}\,\mathcal{F}\,H\,Q$". Another useful property of the predicate local is its idempotence: for any predicate $\mathcal{F}$, the predicate "$\mathsf{local}\,\mathcal{F}$" is equivalent to the predicate "$\mathsf{local}\,(\mathsf{local}\,\mathcal{F})$". In order to conveniently exploit the idempotence property, we introduce a predicate, called $\mathsf{islocal}\,\mathcal{F}$, which asserts that the predicate $\mathcal{F}$ is extensionally equivalent to "$\mathsf{local}\,\mathcal{F}$".

$$\mathsf{islocal}\,\mathcal{F} \quad \equiv \quad (\mathcal{F} = \mathsf{local}\,\mathcal{F})$$

A formula satisfying $\mathsf{islocal}$ is called a *local* formula. As expected, we can prove that "$\mathsf{islocal}\,(\mathsf{local}\,\mathcal{F})$" is true for any predicate $\mathcal{F}$ of the appropriate type. In particular, a characteristic formula is always a local formula. The interest of introducing the predicates $\mathsf{islocal}$ is that it conveniently allows us to apply any

$$
\begin{array}{lll}
[\![\hat{v}]\!] & \equiv & \mathsf{local}\,(\lambda HQ.\ H \rhd Q\,\lceil \hat{v} \rceil) \\
[\![\hat{v}_1\,\hat{v}_2]\!] & \equiv & \mathsf{local}\,(\lambda HQ.\ \mathsf{App}\,\lceil \hat{v}_1 \rceil\,\lceil \hat{v}_2 \rceil\,H\,Q) \\
[\![\mathsf{let}\,x = \hat{t}_1\,\mathsf{in}\,\hat{t}_2]\!] & \equiv & \mathsf{local}\,(\lambda HQ.\ \exists Q'.\ [\![\hat{t}_1]\!]\,H\,Q' \wedge\ \forall x.\ [\![\hat{t}_2]\!]\,(Q'\,x)\,Q) \\
[\![\hat{t}_1\,;\,\hat{t}_2]\!] & \equiv & \mathsf{local}\,(\lambda HQ.\ \exists Q'.\ [\![\hat{t}_1]\!]\,H\,Q' \wedge [\![\hat{t}_2]\!]\,(Q'\,tt)\,Q) \\
[\![\mathsf{let\,rec}\,f = \varLambda\overline{A}.\lambda x.\hat{t}_1\ \mathsf{in}\,\hat{t}_2]\!] & \equiv & \mathsf{local}\,(\lambda HQ.\ \forall f.\ \mathcal{H} \Rightarrow [\![\hat{t}_2]\!]\,H\,Q) \\
 & & \quad \text{with } \mathcal{H} \equiv \forall \overline{A} x H' Q'.\ [\![\hat{t}_1]\!]\,H'\,Q' \Rightarrow \mathsf{App}\,f\,x\,H'\,Q' \\
[\![\mathsf{let}\,x = \varLambda\overline{A}.\,\hat{v}\,\mathsf{in}\,\hat{t}]\!] & \equiv & \mathsf{local}\,(\lambda HQ.\ \forall x.\ x = \lambda\overline{A}.\lceil \hat{v} \rceil \Rightarrow [\![\hat{t}]\!]\,H\,Q) \\
[\![\mathsf{assert\ false}]\!] & \equiv & \mathsf{local}\,(\lambda HQ.\ \mathsf{False}) \\
[\![\mathsf{assert}\ \hat{t}]\!] & \equiv & \mathsf{local}\,(\lambda HQ.\ [\![\hat{t}]\!]\,H\,(\lambda x.\,[x = \mathsf{true}] * H)\ \wedge\ H \rhd Q\,tt) \\
[\![\mathsf{if}\,\hat{v}\,\mathsf{then}\,\hat{t}_1\,\mathsf{else}\,\hat{t}_2]\!] & \equiv & \mathsf{local}\,(\lambda HQ.\quad (\lceil \hat{v} \rceil = \mathsf{true} \Rightarrow [\![\hat{t}_1]\!]\,H\,Q) \\
 & & \qquad\qquad \wedge\, (\lceil \hat{v} \rceil = \mathsf{false} \Rightarrow [\![\hat{t}_2]\!]\,H\,Q)) \\
[\![\mathsf{match}\,\hat{v}\,\mathsf{with}\,\emptyset]\!] & \equiv & \mathsf{local}\,(\lambda HQ.\ \mathsf{False}) \\
[\![\mathsf{match}\,\hat{v}\,\mathsf{with}\,\hat{p} \mapsto \hat{t}\,|\,\hat{b}]\!] & \equiv & \mathsf{local}\,(\lambda HQ.\quad \big(\forall \overline{x}.\ \lceil \hat{v} \rceil = \lceil \hat{p} \rceil \Rightarrow [\![\hat{t}]\!]\,H\,Q\big) \\
 & & \qquad\qquad \wedge\, \big((\forall \overline{x}.\ \lceil v \rceil \neq \lceil \hat{p} \rceil) \Rightarrow [\![\mathsf{match}\,\hat{v}\,\mathsf{with}\,\hat{b}]\!]\,H\,Q\big) \\
 & & \quad \text{where } \overline{x} \text{ are the free variables of } \hat{p}.
\end{array}
$$

Fig. 2. Generation of characteristic formulae.

of the reasoning rules, an arbitrary number of times, and in any order. When reasoning about characteristic formulae, we never unfold the definition of local or islocal, but instead systematically use one of the high-level lemmas shown below.

$$
\begin{array}{ll}
\text{FRAME:} & \mathsf{islocal}\,\mathcal{F}\ \wedge\ \mathcal{F}\,H\,Q\ \Rightarrow\ \mathcal{F}\,(H * H')\,(Q \star H') \\
\text{GC-PRE:} & \mathsf{islocal}\,\mathcal{F}\ \wedge\ \mathcal{F}\,H\,Q\ \Rightarrow\ \mathcal{F}\,(H * H')\,Q \\
\text{GC-POST:} & \mathsf{islocal}\,\mathcal{F}\ \wedge\ \mathcal{F}\,H\,(Q \star H')\ \Rightarrow\ \mathcal{F}\,H\,Q \\
\text{CONSEQUENCE-PRE:} & \mathsf{islocal}\,\mathcal{F}\ \wedge\ \mathcal{F}\,H\,Q\ \wedge\ H' \rhd H\ \Rightarrow\ \mathcal{F}\,H'\,Q \\
\text{CONSEQUENCE-POST:} & \mathsf{islocal}\,\mathcal{F}\ \wedge\ \mathcal{F}\,H\,Q\ \wedge\ Q \blacktriangleright Q'\ \Rightarrow\ \mathcal{F}\,H\,Q' \\
\text{EXTRACT-PROP:} & \mathsf{islocal}\,\mathcal{F}\ \wedge\ (\mathcal{P} \Rightarrow \mathcal{F}\,H\,Q)\ \Rightarrow\ \mathcal{F}\,([\mathcal{P}] * H)\,Q \\
\text{EXTRACT-EXISTS:} & \mathsf{islocal}\,\mathcal{F}\ \wedge\ (\forall x.\ \mathcal{F}\,H\,Q)\ \Rightarrow\ \mathcal{F}\,(\exists x.\,H)\,Q
\end{array}
$$

Note that the predicate islocal also plays a key role in the characteristic formulae of for-loops and while-loops, as we will see in §3.10.

3.7 Construction of characteristic formulae

The characteristic formula of a typed term $\hat{t}$ is written $[\![\hat{t}]\!]$. If $\hat{t}$ admits the weak-ML type T, then the formula $[\![\hat{t}]\!]$ has type $\mathsf{Hprop} \to ([\![T]\!] \to \mathsf{Hprop}) \to \mathsf{Prop}$, where Hprop stands for $\mathsf{Heap} \to \mathsf{Prop}$. The rules for constructing characteristic formulae appear in Figure 2. Observe that every definition starts with an application of the predicate local, and that all the Caml values are translated into Coq values using the decoding operator $\lceil \hat{v} \rceil$.

The first rule from Figure 2 states that a value v admits a pre-condition H and a post-condition Q if the current heap H, also satisfies the predicate $Q\,\lceil \hat{v} \rceil$. The same rule applies in particular to variables. The treatment of applications, function definitions, sequences and let-bindings has already been explained in the introduction.

A polymorphic function is written "$\mathsf{let\,rec}\, f = \Lambda\overline{A}.\lambda x.\hat{t}_1$", where $\overline{A}$ denotes the list of type variables involved in the type-checking of the body of the function. The type variables from the list $\overline{A}$ are quantified in the hypothesis $\mathcal{H}$ provided by the characteristic formula for reasoning about the body of the function. Here again, the type variables are assigned the kind Type in Coq. Recall that, in weak-ML, a polymorphic function admits the type func, just like any other function. So, the variable f involved in the characteristic formula admits in Coq the type Func.

Consider now a polymorphic let-binding of the form "$\mathsf{let}\, x = \Lambda\overline{A}.\, \hat{v}\,\mathsf{in}\,\hat{t}$", where $\hat{v}$ is a polymorphic value with free type variables $\overline{A}$. If $\hat{v}$ has type T, then the program variable x has type $\forall\overline{A}.T$. The characteristic formula associated with this let-binding quantifies over a Coq variable x of type $\forall\overline{A}.[\![T]\!]$, and provides the assumption that x is the Coq value that corresponds to the program value $\hat{v}$. This assumption is stated through an extensional equality, written $x = \lambda\overline{A}.\lceil \hat{v} \rceil$. This equality implies that, for any list of weak-ML types $\overline{U}$, the iterated application of x to the types from the list $[\![\overline{U}]\!]$ yields the Coq value that corresponds to the program value $[\overline{A} \to \overline{U}]\,\hat{v}$.

For assertions, we first consider the particular case of "$\mathsf{assert\ false}$". The characteristic formula for such a term requires the programmer to prove that the corresponding point in the code can never be reached. This task is equivalent to proving that the set of assumptions accumulated before reaching the "$\mathsf{assert\ false}$" contains a logical inconsistency, i.e., that False is derivable. For the more general form "$\mathsf{assert}\ \hat{t}$", the characteristic formula requires the term $\hat{t}$ to evaluate to true and to not change the heap in any visible way. Indeed, it is a desirable property that the program remains correct even if we disable the execution of assertions at runtime. Note that this requirement does not preclude the possibility for $\hat{t}$ to perform side-effects, as long as these effects do not alter the execution of the rest of the program.

The characteristic formula for conditionals asserts that, in order to prove that "$\mathsf{if}\,\hat{v}\,\mathsf{then}\,\hat{t}_1\,\mathsf{else}\,\hat{t}_2$" admits a particular specification, we need to prove that $\hat{t}_1$ admits this specification when $\hat{v}$ is true and that $\hat{t}_2$ admits this same specification when $\hat{v}$ is false. The characteristic formulae for pattern matching generalize that of conditionals. Consider a pattern matching whose first branch is of the form $\hat{p} \mapsto \hat{t}$, that is, it involves a pattern $\hat{p}$ and a body $\hat{t}$. The characteristic formula is made of a conjunction of two propositions, corresponding to the two possible cases. On the one hand, if the value $\hat{v}$ being matched is equal to some instantiation of the pattern $\hat{p}$, then we evaluate the body of the branch, $\hat{t}$, with the pattern variable appropriately instantiated. On the other hand, if the argument $\hat{v}$ of the pattern matching does not match the pattern $\hat{p}$, then we consider the characteristic formula of the remaining clauses.

A pattern matching with no branch left is equivalent to "$\mathsf{assert\ false}$". In the particular case where the pattern matching can be proved to be exhaustive (e.g., by the exhaustivity procedure implemented in the Caml compiler), we know that one of the branches must match, so we may safely replace $\mathsf{local}\,(\lambda HQ.\ \mathsf{False})$. with $\mathsf{local}\,(\lambda HQ.\ \mathsf{True})$. Such a change saves the need for the user to prove interactively the exhaustiveness of the pattern matching.

Note that the characteristic formula for pattern matching involves an auxiliary operation for decoding patterns, written $\lceil \hat{p} \rceil$. This operation maps a well-typed pattern $\hat{p}$ of weak-ML type T towards the corresponding Coq value of type $[\![T]\!]$, by replacing all data constructors with their logical counterpart. Note also that the recursive treatment of pattern matching defined in Figure 2 leads to the duplication of the value $\lceil \hat{v} \rceil$, possibly making the size of the formula grow non-linearly. To avoid this duplication, whenever the argument $\hat{v}$ of the pattern matching is not reduced to a variable or to a trivial value, we transform the pattern matching from the form "$\mathsf{match}\,\hat{v}\,\mathsf{with}\,\hat{b}$" to the form "$\mathsf{let}\,x = \hat{v}\,\mathsf{in}\,\mathsf{match}\,x\,\mathsf{with}\,\hat{b}$" before computing the characteristic formula.

This completes the description of Figure 2. The treatment of mutually-recursive functions is a straightforward generalization; it can be found in the author's dissertation [8]. The characteristic formulae for loops are explained in the next section.

For each construction of the programming language, we set up a Coq notation for pretty-printing the characteristic formula in a way that resembles the source code. We have already explained in §2.2 how to pretty-print formulae for sequences and let-bindings. We show below additional examples concerning values, applications, function definitions and pattern matching.

$$
\begin{array}{l}
(\mathbf{ret}\ V) \ \equiv\ \mathsf{local}\,(\lambda HQ.\ H \rhd Q\,V) \\
(\mathbf{app}\ V_1\ V_2) \ \equiv\ \mathsf{local}\,(\lambda HQ.\ \mathsf{App} V_1 V_2 H Q) \\
(\mathbf{let\ rec} f = (\mathbf{fun}\,\overline{A}\,x := \mathcal{F}_1)\ \mathbf{in}\ \mathcal{F}_2) \ \equiv\ \mathsf{local}\,(\lambda HQ. \\
\qquad \forall f.\ (\forall \overline{A} x H' Q'.\ \mathcal{F}_1\,H'\,Q' \Rightarrow \mathsf{App}\,f\,x\,H'\,Q') \Rightarrow \mathcal{F}_2\,H\,Q) \\
(\mathbf{case}\ V\ \mathbf{is}\ V'\ \mathbf{vars}\ \overline{X}\ \mathbf{then}\ \mathcal{F}_1\ \mathbf{else}\ \mathcal{F}_2) \ \equiv\ \mathsf{local}\,(\lambda HQ. \\
\qquad (\forall \overline{X}.\ V = V' \Rightarrow \mathcal{F}_1\,P) \ \wedge\ ((\forall \overline{X}.\ V \neq V') \Rightarrow \mathcal{F}_2\,P))
\end{array}
$$

3.8 Specification of primitive functions

We give below the specification of the primitive functions for manipulating references. All these functions are reflected in Coq as abstract values of type Func. Below, v has type A, v' has type A', and r and r' have type Loc. Observe that the specification of set allows for strong updates, that is, for changes in the type of the content of a reference cell.

$$
\begin{array}{ll}
\forall A\,v. & \mathsf{App\,ref}\,v\,[\,]\,(\lambda r.\ r \hookrightarrow_A v) \\
\forall A\,r\,v. & \mathsf{App\,get}\,r\,(r \hookrightarrow_A v)\,(\lambda x.\ [x = v] * r \hookrightarrow_A v) \\
\forall A\,A'\,r\,v\,v'. & \mathsf{App\,set}\,(r, v)\,(r \hookrightarrow_{A'} v')\,(\lambda\_.\ r \hookrightarrow_A v) \\
\forall r\,r'. & \mathsf{App\,cmp}\,(r, r')\,[\,]\,(\lambda x.\ [x = \mathsf{true} \Leftrightarrow r = r'])
\end{array}
$$

3.9 Treatment of n-ary functions

The treatment of functions that expects several arguments depend on how such functions are represented in the programming language, and on whether partial applications are supported. There are three main cases.

First, consider the case of a language with n-ary applications: functions are applied to list of arguments, in the form $f(v_1, ..., v_n)$. In this case, we generalize the predicate "$\mathsf{App}\, F\, V\, H\, Q$" to the n-ary form "$\mathsf{App}_n\, F\, V_1 \dots V_n\, H\, Q$", and then use this predicate directly in characteristic formulae and specifications.

Second, consider the case of a ML-style language where the user is encouraged to tuple the arguments: an application takes the form of a function being applied to a single value that happens to be a tuple, i.e. $f\ (v_1, ..., v_n)$. In this case, we can specify functions using the predicate App. We define "$\mathsf{App}_n\, F\, V_1 \dots V_n\, H\, Q$" simply as a convenient notation for the predicate "$\mathsf{App}\, F\, (V_1,\, ...,\, V_n)\, H\, Q$". In this setting, it remains possible (although cumbersome) to reason about curried functions, by explicitly describing their behavior. For example, the specification of a curried function of two arguments asserts that the application of this function to a first argument returns a function that still expects one more argument.

Third, consider the case of a language with curried functions such as Caml. Function calls take the form form $f\, v_1 \dots v_n$. Curried functions are delicate to reason about due to the possibility for partial applications and over applications. In particular, we need lemmas and tactics to handle the case where a function specified as curried function of n arguments is being applied to m arguments, possibly with $n \neq m$. Of course, all these lemmas should be applied automatically, so that reasoning about applications be as smooth as possible for the user. To achieve this, we have developed in CFML a predicate that captures the fact that a function is curried and that all its partial applications terminate. When we use this predicate to specify a function, we are able to automatically derive the specification of a regular, partial or over application of this function. Details can be found in [8].

If we were able to design the programming language, we would try to simplify as much as possible the reasoning about applications. We would avoid curried functions and opt for a built-in construct for n-ary applications, of the form $f(v_1, ..., v_n)$. To support partial applications, we would allow the arguments of a function to be underscore symbols, to indicate that these arguments should be re-abstracted on the fly.

3.10 Characteristic formulae for loops

Loops can be encoded as recursive functions. So, from a theoretical perspective, we do note need to give specific characteristic formulae for loops. That said, loops admit direct characteristic formulae whose use greatly shortens verification proof scripts. To understand the characteristic formula of a while loop, it is useful to first study an example.

Consider the term "$\mathsf{while}\, (\mathsf{get}\, r > 0)\, \mathsf{do}\, (\mathsf{decr}\, r\, ;\, \mathsf{incr}\, s)$", and call this term t. Let us prove that, for any non-negative integer n and any integer m, the term t

admits the pre-condition "$(r \mapsto n) * (s \mapsto m)$" and the post-condition "$\lambda\_.\,(r \mapsto 0) * (s \mapsto m+n)$". We can prove this statement by induction on n. According to the semantics of a while loop, the term t admits the same semantics as the term "$\mathsf{if}\,(\mathsf{get}\,r > 0)\,\mathsf{then}\,(\mathsf{decr}\,r\,;\,\mathsf{incr}\,s\,;\,t)\,\mathsf{else}\,tt$". If the content of r is zero, then n is equal to zero, and it is straightforward to check that the pre-condition matches the post-condition. Otherwise, the decrement function and increment function are called, so the state after their execution is described as "$(r \mapsto n-1) * (s \mapsto m+1)$". At this point, we need to reason about the subsequent iterations of the loop. To that end, we invoke the induction hypothesis, which asserts that the term t, under the pre-condition "$(r \mapsto n-1) * (s \mapsto m+1)$", admits the post-condition "$\lambda\_.\,(r \mapsto 0) * (s \mapsto (m+1)+(n-1))$". The latter matches the required post-condition, that is, "$\lambda\_.\,(r \mapsto 0) * (s \mapsto m+n)$".

This example illustrates how the reasoning about a while loop is equivalent to the reasoning about a conditional whose first branch ends with a call to the same while loop. The characteristic formula of "$\mathsf{while}\,t_1\,\mathsf{do}\,t_2$" builds upon this idea. It involves a quantification over an abstract variable R, which denotes the semantics of the while loop, in the sense that $R\,H'\,Q'$ holds if and only if the loop admits H' as pre-condition and Q' as post-condition. The main assumption provided about R states that, in order to establish the proposition $R\,H'\,Q'$ for a particular H' and Q', it suffices to prove that the term "$\mathsf{if}\,t_1\,\mathsf{then}\,(t_2\,;\,\mathsf{while}\,t_1\,\mathsf{do}\,t_2)\,\mathsf{else}\,tt$" admits H' as pre-condition and Q' as post-condition. This latter statement is expressed with help of the pieces of notation introduced for pretty-printing characteristic formulae. The characteristic formula for while loops is therefore as shown below —the role of the hypothesis "$\mathsf{islocal}\,R$" is explained afterwards.

$$\begin{array}{l} [\![\mathsf{while}\,\hat{t}_1\,\mathsf{do}\,\hat{t}_2]\!] \;\equiv\; \mathsf{local}\,(\lambda HQ.\;\forall R.\;\mathsf{islocal}\,R \wedge \mathcal{H} \;\Rightarrow\; R\,H\,Q) \\ \qquad \text{with } \mathcal{H} \;\equiv\; \forall H'Q'.\;(\mathbf{if}\;[\![\hat{t}_1]\!]\;\mathbf{then}\;([\![\hat{t}_2]\!]\,\mathbf{;}\,R)\;\mathbf{else\;ret}\;tt)\,H'\,Q' \;\Rightarrow\; R\,H'\,Q' \end{array}$$

With the characteristic formula shown above, the verification of a while-loop can be conducted by induction on any well-founded relation. We also provide in CFML tactics to address the typical case where the proof is conducted using a loop invariant and a termination measure.

The hypothesis "$\mathsf{islocal}\,R$" reflects the fact that the predicate R supports application of the frame rule as if it were a characteristic formula. For example, this assumption would be useful for reasoning about the traversal of an imperative list using a while-loop. At every iteration of such a loop, one cell is traversed. This cell may be framed out from the reasoning about the subsequent iterations, thanks to the assumption "$\mathsf{islocal}\,R$". Such an application of the frame rule makes it possible to verify a list traversal using only the simple list representation predicate, avoiding the need to involve the list-segment representation predicate. A similar observation about the usefulness of applying the frame rule during the execution of a loop was also made by Tuerk [45].

The characteristic formula of a for-loop is somewhat similar to that of a while-loop. The main difference is that the predicate R is replaced with a predicate S which takes as extra argument the current value of the loop counter. The definition is as follows.

$$[\![\mathsf{for}\, i = \hat{v}_1\, \mathsf{to}\, \hat{v}_2\, \mathsf{do}\, \hat{t}]\!] \;\equiv\; \mathsf{local}\, (\lambda HQ.\, \forall S.\; (\forall i.\, \mathsf{islocal}\, (S\, i)) \wedge \mathcal{H} \Rightarrow S\, \lceil \hat{v}_1 \rceil\, H\, Q)$$
with $\mathcal{H} \equiv \forall i H' Q'.(\mathbf{if}\, i \le \lceil \hat{v}_2 \rceil\; \mathbf{then}\; ([\![\hat{t}]\!]\, \mathbf{;}\, S\, (i+1))\; \mathbf{else\; ret}\; tt)\, H'\, Q' \Rightarrow S\, i\, H'\, Q'$

4 Soundness and completeness

4.1 Soundness theorem

The soundness theorem states that if the characteristic formula of a program holds of some specification, then this program indeed satisfies that specification. More precisely, if the characteristic formula of a term t holds of a pre-condition H and a post-condition Q, then the execution of t, starting from an initial state h_0 satisfying the pre-condition H, terminates and produces a value v in a final state h_f such that the post-condition Q holds of v and h_f. We write $\hat{t}_{/h_0} \Downarrow \hat{v}_{/h_f}$ the evaluation judgment involved here. The formal statement shown below also takes into account the fact the final heap may contain values that have been subject to the garbage-collection reasoning rule. These values are gathered in a sub-heap called h_g. In other words, $h_f + h_g$ corresponds to the full final heap, possibly including values that are unreachable from the perspective of the garbage collector.

Theorem 1 (Soundness). *Let $\hat{t}$ be a well-typed, closed weak-ML term. Let H be a pre-condition, Q a post-condition, and h_0 be a heap.*

$$[\![\, \hat{t}\,]\!]\; H\; Q \;\;\wedge\;\; H\, h_0 \;\;\Rightarrow\;\; \exists \hat{v}\, h_f\, h_g.\; \hat{t}_{/h_0} \Downarrow \hat{v}_{/(h_f+h_g)} \;\;\wedge\;\; Q\, \lceil \hat{v} \rceil\, h_f$$

Above, H has type "$\mathsf{Heap} \to \mathsf{Prop}$" and Q has type "$\lceil\!\lceil T \rceil\!\rceil \to \mathsf{Heap} \to \mathsf{Prop}$", where T is the weak-ML type of $\hat{t}$.

The proof of the soundness theorem is conducted on a slightly more general statement, which takes into account the fact that, due to the possible application of the frame rule, the pre-condition H may describe only some portion of the entire heap h_0. We thus decompose the heap h_0 in two disjoint parts: h_i, which corresponds to the part covered by the pre-condition, and h_k, which corresponds to the part that has been framed out. The disjointness of these two heaps is written $h_i \perp h_k$. Note that the heap h_k remains unaffected during the evaluation of the term. The generalized statement of soundness is stated using an auxiliary predicate called sound, as follows.

$$\forall \hat{t}\, H\, Q.\quad [\![\, \hat{t}\,]\!]\, H\, Q \;\;\Rightarrow\;\; \mathsf{sound}\, \hat{t} H Q$$

where

$$\mathsf{sound}\, \hat{t} H Q \;\;\equiv\;\; \forall h_i\, h_k.\;\; \begin{cases} h_i \perp h_k \\ H\; h_i \end{cases} \;\;\Rightarrow\;\; \exists \hat{v}\, h_f\, h_g.\, \begin{cases} h_f \perp h_g \perp h_k \\ \hat{t}_{/(h_i+h_k)} \Downarrow \hat{v}_{/(h_f+h_g+h_k)} \\ Q\; \lceil \hat{v} \rceil\; h_f \end{cases}$$

One important aspect of the soundness proof is the realization of the abstract type Func and of the abstract predicate App. For these realizations, we refer to a deep embedding of the source programming language in Coq, that is, a description of the syntax and the semantics of the source language using inductive definitions. Let $\mathsf{well\text{-}typed\text{-}closure}$ be a predicate that characterizes values of the form $\mu f.\Lambda\overline{A}.\lambda x.\hat{t}$ that are well-typed in weak-ML. The type Func is constructed as dependent pairs made of (the deep embedding of) a value $\hat{v}$ and of a proof that $\hat{v}$ satisfies the predicate $\mathsf{well\text{-}typed\text{-}closure}$.

$$\mathsf{Func} \quad \equiv \quad \Sigma_{\hat{v}}(\mathsf{well\text{-}typed\text{-}closure}\,\hat{v})$$

For the purpose of the proof, we also extend the decoding operation to weak-ML values of type func, which are mapped to Coq values of type Func. More precisely, a weak-ML closure is mapped to a dependent pair made of (the deep embedding of) this closure and a proof that this closure is well-typed. Formally:

$$\begin{array}{rcl} \lceil \mu f.\Lambda\overline{A}.\lambda x.\hat{t} \rceil & \equiv & (\mu f.\Lambda\overline{A}.\lambda x.\hat{t},\ \mathcal{H}) \ : \mathsf{Func} \\ & & \text{where } \mathcal{H} \text{ is a proof of ``}\mathsf{well\text{-}typed\text{-}closure}\,(\mu f.\Lambda\overline{A}.\lambda x.\hat{t})\text{''} \end{array}$$

The realization of the predicate $\mathsf{App}\,F\,V\,H\,Q$ asserts that F and V correspond to the decoding of a weak-ML function $\hat{f}$ and a weak-ML value $\hat{v}$ such that the application of the $\hat{f}$ to $\hat{v}$ yields a term that admits the pre-condition H and the post-condition Q in the sense of the predicate sound.

$$\mathsf{App}\,F\,V\,H\,Q \quad \equiv \quad \exists \hat{f}\hat{v}.\ F = \lceil \hat{f} \rceil \ \wedge\ V = \lceil \hat{v} \rceil \ \wedge\ \mathsf{sound}\,(\hat{f}\,\hat{v})\,H\,Q$$

The realizations of Func and App play a key role in the justification of the soundness of characteristic formulae for function definitions and function applications. In summary, even though characteristic formulae support reasoning on programs without involving at any time the deep embedding of the source language in the logic, the justification of the soundness of characteristic formulae critically relies on such a deep embedding.

4.2 Completeness theorem

The completeness theorem is a reciprocal to the soundness theorem. It asserts that if a program admits a given specification, then it is possible to establish this specification using only characteristic formulae. This completeness statement is, of course, relative to the expressive power of the logic of Coq. More precisely, the statement of completeness asserts that if we are able to establish, with respect to a deep embedding of the source language in Coq, that a given program terminates and produces a value satisfying a given post-condition, then we are able to prove in Coq that the characteristic formula of this program holds of the post-condition considered.

The general statement of the completeness theorem involves a number of auxiliary definitions, such as the notion of the most-general specification of a value and of a heap, and the notion of typed reduction. We refer to [8] for these definitions. In the present paper, we only describe a specialized version of the completeness theorem that covers the case of an ML program producing an integer result. This simplified statement reads as follows: if t is a closed ML program whose execution produces an integer n, then the characteristic formula of t holds of a pre-condition that characterizes the empty heap and of a post-condition asserting that the output value is exactly equal to n.

Theorem 2 (Completeness —particular case). *Let t be a closed ML term, let n be an integer and let h be a memory state. Then,*

$$t_{/\emptyset} \Downarrow n_{/h} \quad \Rightarrow \quad [\![\, \hat{t} \,]\!]\; [\,]\; (\lambda x.\, [x = n])$$

The completeness theorem is relative to the expressive power of Coq because the hypothesis $t_{/\emptyset} \Downarrow n_{/h}$ corresponds to a proof in Coq about the semantics of the term t with respect to the deep embedding of the source language.

The proofs of the soundness and completeness theorems are quite involved. They amounts to about 30 pages of the author's PhD dissertation [8]. In addition to those paper-and-pencil proofs, we have also considered a simple imperative programming language (including while loops but no functions) and mechanized the theory of characteristic formulae for this language. More precisely, we formalized the syntax and semantics of this language, defined a characteristic formula generator for it, and then proved in Coq that the formulae produced by this generator are both sound and complete.

4.3 Quantification of type variables

To reflect in characteristic formulae the polymorphism occurring in source programs, we have introduced Coq type variables with the kind Type. (In Coq, *kind* is just a synonymous for *type*.) The Coq expert might feel sceptical about the correctness of the use of Type. Indeed, since a weak-ML type variable is intended to be instantiated with a weak-ML type, the corresponding Coq type variable occurring in a characteristic formula should presumably only be instantiated with a Coq type that corresponds to the translation of a weak-ML type, that is, a Coq type of the form $\llbracket T \rrbracket$.

Thus, it seems that we ought to assign type variables the sort RType, defined as the set of all Coq types that belong to the image of $\llbracket \cdot \rrbracket$, that is $\{\, \mathbb{T} : \mathsf{Type} \mid \exists T.\, \mathbb{T} = \llbracket T \rrbracket \,\}$. In practice, we could provide RType as an abstract definition, since the fact that types correspond to reflected types needs not be exploited in proofs. Nevertheless, we found it more convenient to assign type variables the sort Type instead of RType, because Type is the default kind in Coq. Justifying that this change does not harm the soundness of characteristic formulae is not so straightforward. We explain below the key ideas involved in the proof.

Intuitively, a polymorphic program function cannot inspect its argument in any way. Therefore, the value passed as argument to a polymorphic function does not have to be a Caml value strictly speaking: it could be any *object*. In particular, a polymorphic function could perfectly well manipulate a Coq value that does not correspond to any regular program value. We need to formalize this intuition in order to prove the soundness of the use of Type instead of RType.

To that end, we introduce the notion of *exotic values*. Exotic values are used to embed in the source language the Coq values that do not correspond to the decoding of any regular Caml value. We write $\mathsf{exo}\,\mathbb{T}\,V$ an exotic value that carries a Coq value V of type $\mathbb{T}$ ($\mathbb{T}$ denotes a Coq type). We extend the type system accordingly. We let $\mathsf{Exotic}\,\mathbb{T}$ denote the weak-ML type of an exotic value that carries a Coq value of type $\mathbb{T}$.

$$\begin{array}{lcll} \hat{v} & := & \ldots & \mid\ \mathsf{exo}\,\mathbb{T}\,V \\ T & := & \ldots & \mid\ \mathsf{Exotic}\,\mathbb{T} \end{array}$$

Note that a program never creates an exotic value: exotic values are only used to justify that it is correct to replace RType with the strictly-larger kind Type.

With these exotic values, we can hope to extend the decoding operation $\lceil\cdot\rceil$ and use it to set up a bijection between the set of all typed weak-ML values and the set of all Coq values. Yet, in order to set up such a bijection, we need to carefully handle what we call "semi-exotic values". For example, consider two Coq propositions $\mathcal{P}_1$ and $\mathcal{P}_2$, and consider the value $\mathsf{exo}\,(\mathsf{Prop}\times\mathsf{Prop})\,(\mathcal{P}_1,\mathcal{P}_2)$, which is an exotic value that carries a Coq pair of two propositions, and the value $(\mathsf{exo}\,\mathsf{Prop}\,\mathcal{P}_1,\ \mathsf{exo}\,\mathsf{Prop}\,\mathcal{P}_2)$, which is a Caml pair made of two exotic values each of them carrying a proposition. Both these values correspond to the same Coq value, that is, the Coq pair $(\mathcal{P}_1,\mathcal{P}_2)$. This example shows that it is not obvious to obtain a bijection.

In order to obtain a bijection, we restrict exotic values to only carry Coq values whose head constructor does not match any constructor of a Caml value. To formalize this idea, we introduce the notion of *exotic Coq type*. A Coq type $\mathbb{T}$ is said to be exotic, written "$\mathsf{is\text{-}exotic}\,\mathbb{T}$", if the head constructor of $\mathbb{T}$ does not correspond to a type constructor that exists in weak-ML. The corresponding formal definition appears below, where $\mathscr{C}$ denotes the set of type constructors introduced through algebraic data type definitions.

$$\begin{array}{lcl} \mathsf{is\text{-}exotic}\,\mathbb{T} & \equiv & \quad \mathbb{T}\neq\mathbb{Z}\ \wedge\ \mathbb{T}\neq\mathsf{Loc}\ \wedge\ \mathbb{T}\neq\mathsf{Func} \\ & & \wedge\ (\forall C\in\mathscr{C}.\ \forall\overline{\mathbb{T}'}.\ \ \mathbb{T}\neq C\,\overline{\mathbb{T}'}) \\ & & \wedge\ (\forall\mathbb{T}'.\ \mathbb{T}\neq\forall(A:\mathsf{Type}).\mathbb{T}') \end{array}$$

We then consider a typing rule asserting that an exotic value "$\mathsf{exo}\,\mathbb{T}\,V$" admits the weak-ML type "$\mathsf{Exotic}\,\mathbb{T}$" if and only if $\mathsf{is\text{-}exotic}\,\mathbb{T}$ is true.

We extend the translation of weak-ML types and values into Coq in the obvious manner: the weak-ML type $\mathsf{Exotic}\,\mathbb{T}$ is mapped to the Coq type $\mathbb{T}$ and an exotic value carrying a value V is mapped to the Coq value V.

$$\left.\begin{array}{lcl} \llbracket\mathsf{Exotic}\,\mathbb{T}\rrbracket & \equiv & \mathbb{T} \\ \lceil\mathsf{exo}\,\mathbb{T}\,V\rceil & \equiv & V \end{array}\right\}\quad \textit{when "}\mathsf{is\text{-}exotic}\,\mathbb{T}\textit{" holds}$$

Under these extended definitions, we have proved in [8] that the operator $\llbracket\cdot\rrbracket$ yields a bijection between the set of all weak-ML types and the set of all Coq values that admit the type Type, and that the decoding operator $\lceil\cdot\rceil$ yields a bijection between the set of all typed weak-ML values and the set of all Coq values. In particular, it follows that for every Coq type $\mathbb{T}$, there exists a unique weak-ML type T such that $\mathbb{T} = [\![T]\!]$. Therefore, when weak-ML includes exotic values, the sort RType, which is defined as $\{\, \mathbb{T} : \mathsf{Type} \mid \exists T.\, \mathbb{T} = [\![T]\!] \,\}$, is identical to the sort Type. This observation justifies the assignment of the sort Type to type variables in characteristic formulae.

5 Examples

In this section, we describe six examples. The first one shows how to reason about a simple recursive function that performs side effects. The second one illustrates, using Dijsktra's shortest path algorithm, how CFML supports the reasoning about modular code involving complex invariants. The other four examples focus on particularly delicate programs involving imperative first-class functions. More precisely, we describe the formalization of: (1) a counter function with an abstract local state, (2) Reynold's CPS-append function, (3) an iterator on imperative lists, and (4) the generic operator $\mathsf{compose}$.

To describe abstract data types in CFML, we rely on representation predicates. The formula $v \leadsto R\,V$ is a heap predicate that relates the mutable data structure found at location v with the mathematical value V that it represents. Here, R is a representation predicate: it characterizes the relationship between v, V and the piece of memory state spanned by the data structure under consideration. The predicate $v \leadsto R\,V$ is simply defined as $R\,V\,v$, where R can be any predicate of type $A \to B \to \mathsf{Hprop}$. This section contains examples of definition and use of representation predicates.

5.1 A simple recursive function

Consider the recursive function f, shown below. It adds n to the content of a reference cell r by recursively incrementing this reference.

$$\mathsf{let\,rec}\, f\, r\, n \;=\; \mathsf{if}\,(n = 0)\,\mathsf{then}\,(\mathsf{incr}\, r\,;\; f\, r\,(n-1))$$

Observe that the function enters an infinite loop when $n < 0$.

The function f can be specified in CFML as follows.

$$\forall rna.\quad n \geq 0 \;\Rightarrow\quad \mathsf{App}_2\, f\, r\, n\,(r \hookrightarrow a)\,(\lambda\_.\; r \hookrightarrow (a+n))$$

Observe that the behavior of the function remains unspecified when $n < 0$. Note also that we have chosen to place the pure fact $n \geq 0$ outside of the App predicate. This presentation is more practical, because pure facts can be immediately pushed to the proof context. That said, the formulation $\mathsf{App}_2\, f\, r\, n\,([n \geq 0] * (r \hookrightarrow a))\,(\lambda\_.\; r \hookrightarrow (a+n))$ would be logically equivalent.

The specification above can be proved by induction on n. On the one hand, when $n = 0$, the code returns the unit value. According to the definition of characteristic formula for values, we need to prove the heap implication $(r \hookrightarrow a) \rhd (\lambda\_.\; r \hookrightarrow (a+n))\, tt$. We can check that it is valid using the fact that

n is equal to zero. On the other hand, when $n > 0$, we have to show that, starting from the state $r \mapsto a$, the body of the conditional produces the state $r \mapsto (a{+}n)$. The first instruction from the body of the condition is the increment of r, which takes the state from $r \mapsto a$ to $r \mapsto (a + 1)$. The second instruction is the recursive call to f, for which we can invoke the induction hypothesis. This induction hypothesis, with n instantiated as $n - 1$ (the premise $n - 1 \geq 0$ holds because here $n > 0$) and with a instantiated as $a + 1$, asserts that the recursive call to f transforms the state from $r \mapsto (a+1)$, to $r \mapsto ((a+1)+(n-1))$. After simplification, the latter corresponds to the expected final state $r \mapsto (a + n)$. This concludes the proof.

5.2 Dijkstra's shortest path

We next describe the specification and verification of a Dijkstra's shortest path algorithm. The particular version that we consider uses a priority queue that does not support the decrease-key operation. Using such a queue makes the proofs slightly more involved, because the invariants need to account for the fact that the queue may contain superseded values. The algorithm involves three mutable data structures: v, an array of boolean used to mark the nodes whose edges have already been processed (for these nodes, the best distance is already known); b, an array of distances used to store the best known distance for every node (distances may be infinite); and q, a priority queue for efficiently identifying the next node to visit.

The Caml source code is shown in Figure 3. It is organized around a main while-loop. Inside the loop, the higher-order function List.iter is used for traversing an adjacency list. The implementation of the priority queue is left abstract: the source code is implemented as a Caml functor (not shown in Figure 3), whose argument corresponds to a priority queue module. Similarly, the verification script is implemented as a Coq functor, which expects two arguments: a module representing the implementation of the priority queue, and a module representing the proof of correctness of that queue implementation. This strategy allows to achieve modular verification of modular code.

The specification of the function dijkstra states that if g is the location of a data structure that represents a mathematical graph G through adjacency lists, if the edges in G all have nonnegative weight, and if x and y are indices of two nodes from that graph, then the application of the function dijkstra to g, x and y returns a value d that is equal to the length of the shortest path between x and y in the graph G (d may be infinite). The notion of shortest path is captured by the function dist, which is provided by a Coq library on finite graphs. Moreover, the specification asserts that the structure of the graph is not modified by the execution of the function. The specification is shown below.

$$\begin{array}{l} \forall gxyG.\ \mathsf{nonnegative\_edges}\, G \ \wedge\ x \in \mathsf{nodes}\, G \ \wedge\ y \in \mathsf{nodes}\, G \\ \Rightarrow\ \mathsf{App}_3\, \mathsf{dijkstra}\, g\, x\, y\, (g \leadsto \mathsf{GraphAdjList}\, G) \\ \qquad\qquad\qquad (\lambda d.\, [d = \mathsf{dist}\, G\, x\, y] * (g \leadsto \mathsf{GraphAdjList}\, G)) \end{array}$$

In the specification, the heap predicate $g \leadsto \mathsf{GraphAdjList}\,G$ is used to relate a mathematical graph G with its representation as an array of lists of pairs stored in memory at location g. More precisely, this heap predicate asserts that the memory stores at location g an array, which we may describe by a finite map N from integers to lists of pairs of integers. This array is such that x is an index in N if and only if it is the index of a node in G, and such that a pair (y, w) belongs to the list $N(x)$ if and only if the graph G has an edge of weight w between the nodes x and y. The definition of GraphAdjList is formalized as follows.

$$\begin{aligned}\mathsf{GraphAdjList} \equiv \lambda G g.\ \exists N.\ & (g \leadsto \mathsf{Array}\,N)\\ & * [\forall x.\ x \in \mathsf{nodes}\,G \Leftrightarrow x \in \mathsf{dom}\,N]\\ & * [\forall x \in \mathsf{nodes}\,G.\ \forall y w.\ (x, y, w) \in \mathsf{edges}\,G \Leftrightarrow \mathsf{mem}(y, w)\,N(x)]\end{aligned}$$

The invariant of the main loop of Dijkstra's algorithm, written "$\mathsf{hinv}\,V\,B\,Q$" describes the state of the mutable data structures in terms of three mathematical values: V is a finite map describing the content of the array v, B is a finite map describing the array b, and Q is a multiset of pairs describing the priority queue q. We enforce several logical invariants on the values V, B and Q. These invariants are captured by a record of propositions, written "$\mathsf{inv}\,V\,B\,Q$". The definition of this record is not shown here but, for example, the first field of this record ensures that if $V[z]$ contains the value true then $B[z]$ contains exactly the length of the shortest path between the source x and the node z in the graph G.

The heap description specifying the memory state at each iteration of the main loop therefore takes the following form.

$$\begin{aligned}\mathsf{hinv}\,V\,B\,Q \quad \equiv \quad & (g \leadsto \mathsf{GraphAdjList}\,G) * (v \leadsto \mathsf{Array}\,V)\\ & * (b \leadsto \mathsf{Array}\,B) * (q \leadsto \mathsf{Pqueue}\,Q) * [\mathsf{inv}\,V\,B\,Q]\end{aligned}$$

The proof that the function dijkstra satisfies its specification consists of two parts. The first part is concerned with a number of mathematical theorems that justify the method used by Dijkstra's algorithm for computing shortest paths. This part, which amounts to 180 lines of Coq scripts, is totally independent of characteristic formulae and would presumably be needed in any approach to program verification. The second part consists of a single theorem, whose statement is exactly the specification given earlier on, and whose purpose is to establish that the source code correctly implements Dijkstra's algorithm. The proof of this theorem follows the structure of the characteristic formula generated. It therefore also follows the structure of the source code.

The beginning of the proof script for this verification theorem appears in Figure 4. The script contains three kind of tactics. First, x-tactics are used to make progress through the characteristic formula. For example, the tactic `xwhile_inv` is used to provide the loop invariant and the termination relation. Here, termination is justified by a lexicographical order whose first component is the size of the number of node treated (this number increases from zero up to the total number of nodes) and whose second component is the size of the priority queue. Second, the script makes use of general-purpose Coq tactics (all those whose name does not start with the letter "x"), used for example to name variables,

```
let dijsktra g s e =
    let n = Array.length g in
    let b = Array.make n Infinite in
    let v = Array.make n false in
    let q = Pqueue.create() in
    b.(s) <- Finite 0;
    Pqueue.push (s,0) q;
    while not (Pqueue.is_empty q) do
       let (x,dx) = Pqueue.pop q in
       if not v.(x) then begin
          v.(x) <- true;
          let update (y,w) =
             let dy = dx + w in
             if (match b.(y) with | Finite d -> dy < d
                                  | Infinite -> true)
                then (b.(y) <- Finite dy;
                      Pqueue.push (y,dy) q)
             in
          List.iter update g.(x);
       end;
    done;
    b.(e)
```

Fig. 3. Source code for Dijkstra's algorithm.

unfold invariants, and discharge simple side-conditions. Third, the proof script contains invocations of the mathematical theorems justifying that the algorithm effectively computes shortest paths. For example, the script contains a reference to the lemma `inv_start`, which justifies that the loop invariant holds at the first iteration of the loop. Overall, the verification proof contains a total of 48 lines, including 8 lines of statement of the invariants, and Coq is able to verify the proof in 8 seconds on a 3 GHz machine. These proofs could be discharged even faster if we were to re-implement our tactic for simplifying heap implications directly in Caml.

Figure 5 gives an example of a proof obligation that arises during the verification of the function dijkstra. The set of hypotheses appears above the dashed line. Observe that all the hypotheses are short and well-named. Their names are provided explicitly in the proof script. Providing names is not mandatory, however it generally helps to increase readability of proof obligations and robustness of proof scripts. The proof obligation appears below the dashed line. It consists of a characteristic formula applied to a pre-condition and to a post-condition. Characteristic formulae are pretty-printed in CFML using capitalized keywords (instead of bold keywords) and the sequence operator is written "`;;`".

```
xcf. introv Pos Ns De. unfold GraphAdjList at 1.
hdata_simpl. xextract as N Neg Adj. xapp. intros Ln.
rewrite <- Ln in Neg. xapps. xapps. xapps. xapps ~.
xapps. set (data := fun B V Q => g ~> Array N \*
  v ~> Array V \* b ~> Array B \* q ~> Heap Q).
set (hinv := fun VQ => let '(V,Q) := VQ in Hexists B,
  data B V Q \* [inv G n s V B Q (crossing G s V)]).
xseq (fun _ => Hexists V, hinv (V,\ )).
set (W := lexico2 (binary_map (count (= true)) (upto n))
                 (binary_map card (downto 0))).
xwhile_inv W hinv.
(* -- initial state satisfies the invariant -- *)
refine (ex_intro' (_,_)). unfold hinv,data. hsimpl.
 applys_eq ~ inv_start 2. permut_simpl.
(* -- verification of the loop -- *)
intros [V Q]. unfold hinv. xextract as B Inv. xwhilebody.
...
```

Fig. 4. Beginning of the proof script for Dijkstra's algorithm.

5.3 Counter function

The example of the counter function illustrates the treatment of functions associated with an abstract local state. A counter function is a function that, every time it is called, returns the successor of the integer that it returned on the previous call.

The function create, whose definition is shown below, constructs a new counter function. It allocates a fresh reference r with initial content 0, and then builds a function whose body increments r and returns its content.

$$\mathsf{create} \quad\equiv\quad \lambda\_\,.\,\mathsf{let}\,r = \mathsf{ref}\,0\,\mathsf{in}\,(\lambda\_\,.\,(\mathsf{incr}\,r\,;\,\mathsf{get}\,r))$$

To specify the function create without exposing the fact that its implementation is made of a reference cell, we use a representation predicate, called Cntr. The heap predicate "$f \leadsto \mathsf{Cntr}\,n$" asserts that f is a counter function whose last call returned the value n. The definition of Cntr involves an existential quantification over a predicate I of type "$\mathsf{int} \to \mathsf{Hprop}$" for abstracting over the internal state. More precisely, the existential quantification of I allows us to state that a call to the counter function f takes the counter from a state "$I\,m$" to a state "$I\,(m+1)$" and returns the value $m+1$, without revealing any details of the implementation of this counter function.

$$\mathsf{Cntr}\,n\,f \;\;\equiv\;\; \exists I.\,(I\,n) * [\forall m.\, \mathsf{App}_1\,f\,\mathit{tt}\,(I\,m)\,(\lambda x.\,[x = m+1] * I\,(m+1))]$$

The function create is then specified as producing a new counter f with internal state 0.

$$\mathsf{App}\,\mathsf{create}\,\mathit{tt}\,[\,]\,(\lambda f.\,f \leadsto \mathsf{Cntr}\,0)$$

```
Pos : nonnegative_edges G
Ns : s \in nodes G
Ne : e \in nodes G
Neg : nodes_index G n
Adj : forall x y w : int, x \in nodes G ->
      Mem (y, w) (N\(x)) = has_edge G x y w
Nx : x \in nodes G
Vx : ~ V\(x)
Dx : Finite dx = dist G s x
Inv : inv G n s V' B Q (new_crossing G s x L' V)
EQ : N\(x) = rev L' ++ (y, w) :: L
Ew : has_edge G x y w
Ny : y \in nodes G
______________________________________________(1/6)
(Let dy := Ret (dx + w) in
  Let fy := App ml_array_get b y ; in
   If Match
       (Case fy = Finite d [d] Then Ret (dy < d) Else
       (Case fy = Infinite Then Ret true Else Done))
   Then (App ml_array_set b y (Finite dy) ;) ;;
         App push (y, dy) h ; Else (Ret tt))
(q ~> Pqueue Q \* b ~> Array B \*
  v ~> Array V' \* g ~> Array N)
(fun _:unit => hinv' L)
```

Fig. 5. A proof obligation from the verification of Dijkstra's algorithm.

By unfolding the definition of Cntr, we are able to reason about calls to the function produced by create. That said, we can go even further in terms of abstraction, and present Cntr as an abstract predicate to the client. To that end, it suffices to provide the user with a lemma for reasoning about calls to counter functions. This lemma, shown below, asserts that, in a state $f \rightsquigarrow \mathsf{Cntr}\, n$, a call to f returns the value $n+1$ and updates the state to $f \rightsquigarrow \mathsf{Cntr}\,(n+1)$.

$$\forall f n.\ \mathsf{App}\, f\, tt\, (f \rightsquigarrow \mathsf{Cntr}\, n)\, (\lambda x.\, [x = n+1] * f \rightsquigarrow \mathsf{Cntr}\,(n+1))$$

In summary, the counter function example illustrates how the abstract local state of a function can be entirely packed into a representation predicate.

5.4 Continuations

The CPS-append function has been proposed as a verification challenge by Reynolds [43], for testing the ability to specify and reason about continuations that are used in a nontrivial way. The CPS-append function takes as an argument two lists x and y, as well as an initial continuation k. The function ultimately calls the continuation k on the concatenation of this lists x and y. What makes this function nontrivial is that it does not build the list x++y explicitly. Instead, the function calls itself recursively using a different continuation at every iteration. The nested execution of these continuations starts from the list y and eventually produces the list $x{+\!\!+}y$. This list is then passed as an argument to the original continuation k. The code of the CPS-append function appears below.

```
let rec cpsapp (x y:'a list) (k:'a list->'b) : 'b =
   match x with
   | [] -> k y
   | v::x' -> cpsapp x' y (fun z -> k (v::z))
```

The CPS-append function can be specified as shown below, where k has type Func, x and y have type $\mathsf{list}A$, and ${+\!\!+}$ denotes the Coq concatenation operator.

$$\forall AxykHQ. \quad \mathsf{App}_1\, k\, (x{+\!\!+}y)\, H\, Q \;\Rightarrow\; \mathsf{App}_3\, \mathsf{cpsapp}\, x\, y\, k\, H\, Q$$

Slightly more challenging is the verification of the imperative counterpart of the CPS-append function (whose code is not shown here). It is based on the same principle as the purely-functional version, except that x and y are now pointers to mutable lists and that the continuations mutate pointers in the list x in order to build the concatenation of the two lists in place. The specification of this imperative version appears below, where Mlist is the representation predicate for mutable lists.

$$\begin{aligned}\forall AxykLMHQ. \quad & (\forall z.\ \mathsf{App}_1\, k\, z\, (H * (z \leadsto \mathsf{Mlist}\,(L{+\!\!+}M)))\, Q) \\ & \Rightarrow \mathsf{App}_3\, \mathsf{cpsapp'}\, x\, y\, k\, (H * (x \leadsto \mathsf{Mlist}\, L) * (y \leadsto \mathsf{Mlist}\, M))\, Q\end{aligned}$$

Above, the pre-condition asserts that the locations x and y (here of type Loc) correspond to lists called L and M, respectively. The pre-condition also mentions an abstract heap predicate H, which is needed because the frame rule usually does not apply when reasoning about CPS functions. Indeed, the entire heap needs to be passed on to the continuation[3]. The continuation k is ultimately called on a location z that corresponds to the list $L{+\!\!+}M$. The proof that the imperative CPS-append function satisfies its specification is conducted by induction on L. It is only 8 lines long in Coq.

5.5 Imperative list iterator

To specify the iterator on imperative lists in a useful way, we need a generalized version of the representation predicate for lists. So far, we have used heap predicates of the form $m \leadsto \mathsf{Mlist}\, L$. This predicate works well when the values stored in the list are of some base type. However, in general, the values stored in the list need to be described using their own representation predicate. We therefore use a more general "parametric representation predicate", written $\mathsf{Mlistof}\, T$, where T is

[3] Thielecke [44] suggests that answer-type polymorphism could be used to design reasoning rules that would save the need for quantifying over the heap H passed on to the continuation. However, his technique has limitations, in particular it does not support recursion through the store.

the representation predicate for the elements stored in the list. For example, we may use the heap predicate "$m \leadsto \mathsf{Mlistof}\,\mathsf{Cntr}\,L$" to describe a mutable list that starts at location m and contains a list of disjoint counter functions whose internal states are described by the integer values from the Coq list L. Note that the predicate Mlist used previously is a particular case of $\mathsf{Mlistof}$: it can be obtained by applying $\mathsf{Mlistof}$ to the identity representation predicate "$\lambda X.\,\lambda x.\,[x = X]$".

We are now ready to describe the specification of an higher-order iterator on mutable lists. This iterator, called iter, is implemented using a while loop which traverse the list until reaching a null pointer. The execution of "$\mathsf{iter}\,f\,m$" results in the function f being applied to all the values stored in the list. This execution may result in two kind of side-effects. First, it may modify the values stored in the list. Second, it may affect the state of other mutable data structures. Thus, the initial state takes the form $H * (m \leadsto \mathsf{Mlistof}\,T\,L)$, and the final state takes the form $H' * (m \leadsto \mathsf{Mlistof}\,T\,L')$, where H and H' are two heap descriptions and L and L' are two Coq lists. To introduce some abstraction, we use a predicate called I. The intention is that the proposition $I\,L\,L'\,H\,H'$ captures the fact that, for any m, the term "$\mathsf{iter}\,f\,m$" admits the pre-condition $H * (m \leadsto \mathsf{Mlistof}\,T\,L)$ and the post-condition $\lambda\_.\,(H' * (m \leadsto \mathsf{Mlistof}\,T\,L'))$.

Two assumptions are provided for reasoning about the predicate I. The first one concerns the case where the list is empty. In this case, both L and L' are empty, and H' must match H. The second assumption concerns the case where the list is not empty. In this case, we first perform a call to f and then recursively call the function iter. The initial state of the list is then of the form $X :: L$ and the final state of the form $X' :: L'$. The values X and X' are related by the specification of the function f. This specification also relates the input state H with an intermediate state H'', which corresponds to the state after the call to f and before the recursive call to iter. The formal statement of the assumptions about I are thus as follows.

$$\begin{array}{lll}
\mathcal{H}_1 & \equiv \forall H. & I\,\mathsf{nil}\,\mathsf{nil}\,H\,H \\
\mathcal{H}_2 & \equiv \forall X X' L L' H H' H''. & (\forall x.\ \mathsf{App}_1\,f\,x\,(H * x \leadsto T\,X)\,(H'' * x \leadsto T\,X')) \\
& & \wedge\ \ I\,L\,L'\,H''\,H' \\
& & \Rightarrow\ \ I\,(X :: L)\,(X' :: L')\,H\,H'
\end{array}$$

Above, L and L' have type $\mathsf{list}A$, f has type Func, X has type A, x has type B, and T has type $A \to B \to \mathsf{Hprop}$.

To establish that the term "$\mathsf{iter}\,f\,m$" admits the pre-condition $H * (m \leadsto \mathsf{Mlistof}\,T\,L)$ and the post-condition $\lambda\_.\ H' * (m \leadsto \mathsf{Mlistof}\,T\,L')$, it suffices to prove the proposition $I\,L\,L'\,H\,H'$, where I is the abstract predicate for which only the assumptions $\mathcal{H}_1$ and $\mathcal{H}_2$ are provided. The specification of iter is thus as follows.

$$\begin{array}{l}
\forall A B T f m L L' H H'.\ (\forall I.\,\mathcal{H}_1 \wedge \mathcal{H}_2 \Rightarrow\ I\,L\,L'\,H\,H')\ \Rightarrow \\
\quad \mathsf{App}_2\,\mathsf{iter}\,f\,m\,(H * (m \leadsto \mathsf{Mlistof}\,T\,L))\,(\lambda\_.\ H' * (m \leadsto \mathsf{Mlistof}\,T\,L'))
\end{array}$$

To check the usability of this specification, we describe an example, which involves a list m of distinct counter functions (recall §5.3). The idea is to make a call to each of these counters in order to increment their internal states. The values returned by these calls are simply ignored. What matters here is that every counter sees its current state incremented by one. The function steps implements this scenario.

$$\mathsf{steps} \quad \equiv \quad \lambda m.\ \mathsf{iter}\,(\lambda f.\,\mathsf{ignore}\,(f\ tt))\,m$$

The heap predicate $m \rightsquigarrow \mathsf{Mlistof\,Cntr}\,L$ asserts that the mutable list starting at location m contains a list of counter functions whose internal states are described by the integer values from the Coq list L. A call to the function steps on the list m increments the internal state of every counter, so the final state is described by the heap predicate $m \rightsquigarrow \mathsf{Mlistof\,Cntr}\,L'$, where L' is obtained by adding one to all the elements in L. The specification of the function steps is thus as follows.

$$\forall mL.\ \ \mathsf{App}_1\,\mathsf{steps}\,m\,(m \rightsquigarrow \mathsf{Mlistof\,Cntr}\,L)\,(\lambda\_.\ \ m \rightsquigarrow \mathsf{Mlistof\,Cntr}\,(\mathsf{map}\,(+1)\,L))$$

This example demonstrates the ability of CFML to formally verify the application of a polymorphic higher-order iterator to an imperative list made of first-class functions associated with abstract local state.

5.6 The composition combinator

The function compose is defined as $\lambda g.\,\lambda f.\,\lambda x.\,g\,(f\,x)$. Because this definition is so short and so general, it is hard to imagine a general specification for compose shorter or more abstract that the code itself —unless making specific assumptions about f and g.

With characteristic formulae, we are able to devise a simple, most-general specification for this function. The idea is to use the characteristic formula of its body as specification. Indeed, because characteristic formulae are sound and complete, the characteristic formula of a term is always the most-general specification for this term. The function compose hence admits the specification shown below. (Recall that bold keywords correspond to notation for characteristic formulae.)

$$\begin{aligned} &\forall HQ.\ \ (\mathbf{let}\ y\ =\ \mathbf{app}\ f\ x\ \mathbf{in}\ \mathbf{app}\ g\ y)\,H\,Q \\ &\qquad \Rightarrow\ \mathsf{App}_3\,\mathsf{compose}\,g\,f\,x\,H\,Q \end{aligned}$$

The above specification describes the behavior of compose when applied to three arguments. Nevertheless, the function compose is typically applied to only two arguments. To support this case, the specification can be reformulated as follows.

$$\begin{aligned} &\mathsf{App}_2\,\mathsf{compose}\,g\,f\,[\,]\,(\lambda k. \\ &\quad [\forall xHQ.\ (\mathbf{let}\ y\ =\ \mathbf{app}\ f\ x\ \mathbf{in}\ \mathbf{app}\ g\ y)\,H\,Q \Rightarrow \mathsf{App}_1\,k\,x\,H\,Q]) \end{aligned}$$

In CFML, we are able to automatically derive this latter specification from the earlier one in case of a partial application. Details can be found in the author's PhD dissertation [8].

6 Related work

6.1 Program logics

A program logic consists of a specification language and of a set of reasoning rules that can be used to establish that a program satisfies a specification. Program logics do not directly provide an effective program verification tool, but they may serve as a basis for justifying the correctness of such a tool. Hoare logic [16, 20] is probably the most well-known program logic. Separation Logic [43] is an extension of Hoare logic that supports local reasoning. Separation Logic serves as a basis for a number of verification tools, for example Smallfoot [5]. Separation Logic has also often been used in existing interactive proof assistants, in which Separation Logic formulae can be defined as predicates over heaps, as done, e.g., in [30, 1, 11, 31, 34] as well as in our work.

Dynamic Logic [18] is another program logic. In this modal logic, $\langle t \rangle$ is a modality that embeds a program t in such a way that the formula "$H_1 \rightarrow \langle t \rangle\, H_2$" asserts that, in any heap satisfying H_1, the sequence of commands t terminates and produces a heap satisfying H_2. Dynamic Logic serves as the foundation for the KeY system [4], which targets the verification of Java programs. One limitation of Dynamic Logics is that they depart from standard mathematical logics, precluding the use of standard proof assistants.

Very few program logics support reasoning about higher-order functions. One of them is that developed by Honda, Berger and Yoshida [6]. The specification language of Honda *et al*'s logic is a nonstandard first-order logic, which features an ad-hoc construction, called *evaluation formula*, and written $\{H\}\, v \bullet v' \searrow x\, \{H'\}$. This proposition asserts that, under a heap satisfying H, the application of the value v to the value v' produces a result named x in a heap satisfying H'. This evaluation formula plays a similar role as that of our predicate App. Such evaluation formulae are the key to achieving completeness, because they allow to fully specify the behavior of a function. Another specificity of Honda *et al*'s specification language is that the values of the logic are identified with the values of the programming language, including non-terminating functions. This nonstandard specification language prevented Honda *et al* from building a practical verification tool on top of an existing theorem prover. In contrast, our characteristic formulae are expressed in terms of a standard higher-order logic, making it possible to reuse existing proof tools.

6.2 Verification condition generators

A Verification Condition Generator (VCG) is a tool that, given a program annotated with specifications and invariants, extracts a set of proof obligations. Discharging these proof obligations, either automatically or manually, ensures the correctness of the program. A large number of VCGs targeting various programming languages have been implemented in the last decades. For example, the tool Spec-# [2] parses annotated C# programs, and then produces proof obligations that can then be sent to an SMT solver. Because most SMT solvers are restricted to first-order logic, the specification language is usually restricted to this fragment. Specifications therefore do not benefit from the expressiveness, the modularity, and the elegance of higher-order logic.

A few tools support higher-order logic. One notable example is the tool Why [13], an intermediate language that can be used in conjunction with a front-end, for example Caduceus [14] for C programs or Krakatoa [29] for Java programs. When Why produces a proof obligation that cannot be verified automatically by at an SMT solver, this proof obligation may be discharged using an interactive proof assistant such as Coq. Recent work has focused on trying to extend Why with support for higher-order functions [23], building upon ideas developed for the tool Pangolin [42]. Pangolin's approach consists of representing in the logic a function directly as a pair of a pre-condition and a post-condition, yet this approach precludes the use of auxiliary variables in the specification of first-class functions.

Another tool that supports higher-order logic is Jahob [46], which targets the verification of programs written in a subset of Java. For discharging proof obligations, Jahob relies on a translation from a subset of higher-order logic into first order logic, as well as on automated theorem provers extended with specialized decision procedures for reasoning on lists, trees, sets and maps. A key feature of Jahob is its *integrated proof language*, which allows the user to include proof hints directly inside the source code. Those hints are intended to guide automated theorem provers, in particular by indicating how to instantiate existential variables.

We find that, when it comes to verifying complex programs, it is more practical to work in an interactive environment. Indeed, coming up with the right invariants typically requires a number of iterations. With a VCG tool such as Why or Jahob, if the user changes, say, a local loop invariant, then he needs to run again the VCG tool, wait for the SMT solvers to try and discharge the proof obligations, and then read the remaining obligations, which may have changed quite significantly even upon minor changes by the user. Overall, each cycle is quite time consuming. On the contrary, with characteristic formulae, the user works in an interactive setting that provides nearly-instantaneous feedback on changes: the user can update the invariant and efficiently replay the same proof script, most often without any modification.

6.3 Shallow embeddings

The shallow embedding approach to program verification consists of relating a source program with a corresponding logical definition. The relationship can take three forms.

First, one may write a logical definition and use an extraction mechanism (e.g., [28]) to translate the code into a conventional programming language. For example, Leroy's certified C compiler [26] is developed in this way. However, only purely function code can be produced. Also based on extraction is the tool Ynot [11], which implements Hoare Type Theory (HTT) [37], by axiomatically extending the Coq language with a monad for encapsulating side effects and partial functions. HTT was also later re-implemented by Nanevski *et al* [38] without using any axioms, yet at the expense of loosing the ability to reason on higher-order stores. In HTT, the monad involved has a type of the form "$\mathsf{STsep}\,P\,Q$", and it corresponds to a partial-correctness specification with pre-condition P and post-condition Q. Verification is performed through the type-checking in Coq of the program. This type-checking phase may produce a set of proof obligations that may be discharged interactively using tactics. One advantage of characteristic formulae is that they are able to target an existing programming language, whereas the Ynot programming language has to fit into the logic it is implemented in. For example, supporting features such as alias-patterns and when-clauses would be a real challenge for Ynot, because pattern matching is so deeply hard-wired in Coq that it would be very hard to modify it.

Another technical difficulty faced by HTT is the treatment of auxiliary variables. A specification of the form "$\mathsf{STsep}\,P\,Q$" does not naturally allow for auxiliary variables to be used for sharing information between the pre- and the post-condition. Indeed, if P and Q both refer to a auxiliary variable x quantified outside of the type "$\mathsf{STsep}\,P\,Q$", then x is considered as a computationally-relevant value and thus it will appear in the extracted code. Ynot [11] relies on a hack for simulating the Implicit Calculus of Constructions [3], in which computationally-irrelevant values are tagged explicitly. A danger of this approach is that forgetting to tag a variable as auxiliary lead to the extraction of potentially very inefficient code, because computationally-irrelevant values may get allocated and passed around at runtime.

Other implementations of HTT handle auxiliary variables differently, by relying on post-conditions that refer not only to the output heap but also to the input heap [37, 38]. Such *binary post-conditions* make it possible to eliminate auxiliary variables by duplicating the pre-condition inside the post-condition. For example, "$\forall x.\ \mathsf{STsep}\,P\,Q$" is encoded as "$\mathsf{STsep}\,(\exists x.P)\,(\forall x.\,P \Rightarrow Q)$". HTT [38] then provides tactics to try and avoid the duplication of proof obligations. However, duplication typically remains visible in specifications. This is problematic because specifications are part of the trusted base, so their statement should be as simple as possible.

The second way of relating a source program to a logical definition works in the other direction: it consists of decompiling a piece of conventional source code into a set of logical definitions. This approach is used for example in the LOOP compiler [22] and also in Myreen and Gordon's work [35]. The LOOP compiler takes Java programs and compiles them into PVS definitions. LOOP's proof tactics rely on a weakest-precondition calculus to achieve a high degree of automation. However, interactive proofs require a lot of expertise: LOOP requires the user to understand the compilation scheme involved [22]. By contrast, the tactics manipulating characteristic formulae allow conducting interactive proofs of correctness without detailed knowledge on the construction of those formulae.

Myreen and Gordon showed how to decompile machine code into HOL4 functions [35]. The lemmas proved interactively about the generated HOL4 functions can then be automatically transformed into lemmas about the behavior of the corresponding pieces of machine code. This translation from machine code into HOL4 is only possible because the functional translation of a while loop is a tail-recursive function, and because tail-recursive functions can be accepted as logical definitions in HOL4 without compromising the soundness of the logic, even when the function is non-terminating. Without exploiting this peculiarity of tail-recursive functions, the automated translation of source code into HOL4 would not be possible. For this reason, it does not seem possible to apply this decompilation-based approach to the verification of code featuring general recursion and higher-order functions.

A third approach to using a shallow embedding consists of writing the program to be verified twice, once as a program definition and once as a logical definition, and then proving that the two are related. This approach has been employed in the verification of a microkernel as part of the Sel4 project [25]. Compared with Myreen and Gordon's work [35, 33], the main difference is that the low-level code is not decompiled automatically but instead decompiled by hand. This decompilation phase is then proved correct using semi-automated tactics. The Sel4 approach thus allows for more flexibility in the choice of the logical definitions, yet at the expense of a larger investment from the user. Moreover, like in Myreen and Gordon's work, general recursion is problematic: all the code of the Sel4 microkernel written in the shallow embedding had to avoid any form of nontrivial recursion [24].

In summary, all the approaches based on a shallow embedding share one central difficulty: the need to overcome the discrepancies between the programming language and the logical language, in particular with respect to the treatment of imperative functions, partial functions, and recursive functions. In contrast, characteristic formulae rely on the first-order data type Func for representing functions. As established by the completeness theorem, this approach supports reasoning about all forms of first-class functions.

6.4 Deep embeddings

A deep embedding consists of a description of the syntax and the semantics of a programming language in the logic of a proof assistant, using inductive definitions. In theory, a deep embedding can be used to verify programs written in any programming language, without any restrictions in terms of expressiveness, apart from those of the proof assistant. Mehta and Nipkow [32] have set up the first proof-of-concept by formalizing a basic procedural language in Isabelle/HOL and proving Hoare-style reasoning rules correct with respect to the semantics of that language. More recently, Shao *et al* have developed frameworks such as XCAP [39] for reasoning in Coq about short but complex assembly routines.

In previous work [7], the author has worked on a deep embedding of the pure fragment of Caml inside the Coq proof assistant. Working with deep embedding suffers from at least two major issues. First, deep embeddings involve an explicit representation of syntax. In particular, we need to represent binders, which are notoriously problematic: a representation based on names does not enjoy alpha-conversion, and a representation involving de Bruijn indices makes programs unreadable (unless having dedicated support from the prover). Second, deep embeddings describe values through an encoding. For example, the list of integers "3 :: 2 :: nil" is represented in the deep embedding as the term "$\mathsf{vconstr}_2$ cons (vint 3) ($\mathsf{vconstr}_2$ cons (vint 2) ($\mathsf{vconstr}_0$ nil))" where vconstr is the constructor from the grammar of values used to represent the application of data constructors (the index corresponds to the arity), and where vint is the constructor from the grammar of values used to represent program integers.

This work on a deep embedding [7] then did lead to the development of characteristic formulae, which can be viewed as an abstract layer built on top of a deep embedding: characteristic formulae hide the technical details associated with the explicit representation of syntax while retaining the high expressiveness of that approach. In particular, characteristic formulae avoid the representation of program syntax and directly lift pure program values at the logical level.

7 Conclusion

In this paper, we have explained how to build characteristic formulae for imperative programs. The key ingredients involved in the approach are: (1) a reflection of program values in the logic, using, in particular, the abstract data type Func to represent functions and the abstract predicate App to specify them, (2) a shallow embedding of Separation Logic formulae for describing imperative data structures, (3) a compositional algorithm for constructing a predicate that describes the semantics of the program in the logic, (4) an integration of the frame rule, the rule of consequence and the rule of garbage collection through the predicate local, (5) a notation layer for displaying formulae in a way that resembles the source code, (6) a set of tactics for manipulating characteristic formulae without having to understand how they are built. We have proved that program verification through characteristic formulae is only limited by the expressiveness of the theorem prover. Moreover, we have shown that characteristic formulae can be used to verify nontrivial Caml programs in practice.

Acknowledgments I wish to thank François Pottier for his very useful guidance during my PhD. I also thank the anonymous reviewers for their helpful comments.

References

1. Andrew W. Appel. Tactics for separation logic. *Unpublished draft, http://www.cs.princeton.edu/appel/papers/septacs.pdf*, 2006.
2. Mike Barnett, Rob DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6), 2004.

3. Bruno Barras and Bruno Bernardo. The implicit calculus of constructions as a programming language with dependent types. In *FoSSaCS*, volume 4962 of *LNCS*, pages 365–379. Springer, 2008.
4. Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer-Verlag, Berlin, 2007.
5. Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *International Symposium on Formal Methods for Components and Objects*, volume 4111 of *LNCS*, pages 115–137. Springer, 2005.
6. Martin Berger, Kohei Honda, and Nobuko Yoshida. A logical analysis of aliasing in imperative higher-order functions. In *ICFP*, pages 280–293, 2005.
7. Arthur Charguéraud. Verification of call-by-value functional programs through a deep embedding. 2009. Unpublished. http://arthur.chargueraud.org/research/2009/deep/.
8. Arthur Charguéraud. *Characteristic Formulae for Mechanized Program Verification*. PhD thesis, Université Paris-Diderot, 2010.
9. Arthur Charguéraud. Program verification through characteristic formulae. In *ICFP*, pages 321–332. ACM, 2010.
10. Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceeding of the 16th ACM SIGPLAN International Conference on Functional programming (ICFP)*, pages 418–430. ACM, 2011.
11. Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective interactive proofs for higher-order imperative programs. In *ICFP*, 2009.
12. The Coq Development Team. *The Coq Proof Assistant Reference Manual, Version 8.2*, 2009.
13. Jean-Christophe Filliâtre. Verification of non-functional programs using interpretations in type theory. *Journal of Functional Programming*, 13(4):709–745, 2003.
14. 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, 2004.
15. Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *PLDI*, pages 237–247, 1993.
16. R. W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, 1967.
17. Susanne Graf and Joseph Sifakis. A modal characterization of observational congruence on finite terms of CCS. *Information and Control*, 68(1-3):125–145, 1986.
18. David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. The MIT Press, Cambridge, Massachusetts, 2000.
19. Matthew Hennessy and Robin Milner. On observing nondeterminism and concurrency. In *ICALP*, volume 85 of *LNCS*, pages 299–309. Springer-Verlag, 1980.
20. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, 1969.
21. Kohei Honda, Martin Berger, and Nobuko Yoshida. Descriptive and relative completeness of logics for higher-order functions. In *ICALP*, volume 4052 of *LNCS*. Springer, 2006.
22. Bart Jacobs and Erik Poll. Java program verification at nijmegen: Developments and perspective. In *ISSS*, volume 3233 of *LNCS*, pages 134–153. Springer, 2003.

23. 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. ACM, 2009.
24. Gerwin Klein, Philip Derrin, and Kevin Elphinstone. Experience report: seL4: formally verifying a high-performance microkernel. In *ICFP*, pages 91–96. ACM, 2009.
25. Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd Symposium on Operating Systems Principles (SOSP), Operating Systems Review (OSR)*, pages 207–220, Big Sky, MT, 2009. ACM SIGOPS.
26. Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL*, pages 42–54, 2006.
27. Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system*, 2005.
28. Pierre Letouzey. *Programmation fonctionnelle certifiée – l'extraction de programmes dans l'assistant Coq.* PhD thesis, Université Paris 11, 2004.
29. Claude Marché, Christine Paulin Mohring, and Xavier Urbain. The Krakatoa tool for certification of Java/JavaCard programs annotated in JML. *Journal of Logic and Algebraic Programming (JLAP)*, 58(1–2):89–106, 2004.
30. Nicolas Marti, Reynald Affeldt, and Akinori Yonezawa. Towards formal verification of memory properties using separation logic, 2005.
31. Andrew McCreight. Practical tactics for separation logic. In *TPHOLs*, volume 5674 of *LNCS*, pages 343–358. Springer, 2009.
32. Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. *Information and Computation*, 199(1–2), 2005.
33. Magnus O. Myreen. *Formal Verification of Machine-Code Programs.* PhD thesis, University of Cambridge, 2008.
34. Magnus O. Myreen. Separation logic adapted for proofs by rewriting. In *Interactive Theorem Proving (ITP)*, volume 6172 of *LNCS*, pages 485–489. Springer, 2010.
35. Magnus O. Myreen and Michael J. C. Gordon. Verified LISP implementations on ARM, x86 and powerPC. In *TPHOLs*, volume 5674 of *LNCS*, pages 359–374. Springer, 2009.
36. Aleksandar Nanevski and Greg Morrisett. Dependent type theory of stateful higher-order functions. Technical Report TR-24-05, Harvard University, 2005.
37. Aleksandar Nanevski, J. Gregory Morrisett, and Lars Birkedal. Hoare type theory, polymorphism and separation. *Journal of Functional Programming*, 18(5-6):865–911, 2008.
38. Aleksandar Nanevski, Viktor Vafeiadis, and Josh Berdine. Structuring the verification of heap-manipulating programs. In *POPL*, pages 261–274. ACM, 2010.
39. Zhaozhong Ni and Zhong Shao. Certified assembly programming with embedded code pointers. In *POPL*, 2006.
40. Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL*, volume 2142 of *LNCS*, pages 1–19, Berlin, 2001. Springer-Verlag.
41. Chris Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 1999.
42. Yann Régis-Gianas and François Pottier. A Hoare logic for call-by-value functional programs. In *MPC*, 2008.

43. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.
44. Hayo Thielecke. Frame rules from answer types for code pointers. In *POPL*, pages 309–319, 2006.
45. Thomas Tuerk. Local reasoning about while-loops. In *VSTTE LNCS*, 2010.
46. Karen Zee, Viktor Kuncak, and Martin C. Rinard. An integrated proof language for imperative programs. In *PLDI*, pages 338–351. ACM, 2009.