Characteristic Formulae for the Verification of Imperative Programs

Arthur Charguéraud
Max Planck Institute for Software Systems (MPI-SWS)
charguer@mpi-sws.org

Abstract

In previous work, we introduced an approach to program verification based on characteristic formulae. The approach consists of generating a higher-order logic formula from the source code of a program. This characteristic formula is constructed in such a way that it gives a sound and complete description of the semantics of that program. The formula can thus be exploited in an interactive proof assistant to formally verify that the program satisfies a particular specification.

This previous work was, however, only concerned with purely-functional programs. In the present paper, we describe the generalization of characteristic formulae to an imperative programming language. In this setting, characteristic formulae involve specifications expressed in the style of Separation Logic. They also integrate the frame rule, which enables local reasoning. We have implemented a tool based on characteristic formulae. This tool, called CFML, 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.

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

General Terms Verification

1. Introduction

This paper addresses the problem of building formal proofs of correctness for higher-order imperative programs. It describes an effective technique for verifying that a program satisfies a specification, and for proving termination of that program. This technique supports the verification of arbitrarily-complex properties, thanks to the use of an interactive proof assistant based on higher-order logic. The work described in this paper is based on the notion of *characteristic formula* of a program. A characteristic formula is a higher-order logic formula that fully characterizes the semantics of a program, and may thus be used to prove properties about the behavior of that program.

In previous work, we have shown how to build and exploit characteristic formulae for purely-functional programs [9]. In this paper, we extend those results to an imperative programming language. Let $[\![t]\!]$ denote the characteristic formula of a imperative term t. The application of the predicate $[\![t]\!]$ to a pre-condition H and to a post-condition Q yields the proposition $[\![t]\!]\ H\ Q$. By construction of characteristic formulae, this proposition is true if and only if the term t admits H as pre-condition and Q as post-condition. The proposition $[\![t]\!]\ H\ Q$ may be established through interactive proofs, using a combination of general-purpose tactics and tactics specialized for the manipulation of characteristic formulae.

Characteristic formulae are designed to be easily readable and easily manipulable from inside an interactive proof assistant. A characteristic formula has a size linear in that of the program it describes. Moreover, a characteristic formula can be displayed in a way that closely resembles the source code that it describes, thanks to the use of an appropriate system of notation. With this notation system, the proof obligation $[\![t]\!]\ H\ Q$ stating that "the term t admits H as pre-condition and Q as post-condition" is displayed in a way that reads as "$t\ H\ Q$". This display feature makes it easy to relate proof obligations to the piece of code they arose from.

The notion of characteristic formulae originates in process calculi. In this context, two processes are behaviorally equivalent if and only if their characteristic formulae are logically equivalent [16]. An algorithm for building the characteristic formula of any process was proposed in the 80's [14]. More recently, Honda, Berger and Yoshida adapted this idea from process logics to program logics [18]. 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. Since the construction of a theorem prover dedicated to this logic would have required a tremendous effort, Honda *et al*'s work remained theoretical and did not result in an effective program verification tool.

In prior work [9], we showed how to construct characteristic formulae that are expressed in a standard higher-order logic. Moreover, we showed that characteristic formulae can be made of linear size and that they can be pretty-printed like the source code they describe. Those formulae are therefore suitable for manipulation inside an existing proof assistant such as Coq [11]. We have implemented a tool, called CFML (short for *Characteristic Formulae for ML*) that parses a Caml program [24] and produces its characteristic formula in the form of a Coq statement. Using CFML, we were able to verify more than half of the content of Okasaki's reference book *Purely Functional Data Structures* [37]. Since then, we have generalized characteristic formulae to support reasoning about mutable state, and have updated CFML accordingly. In the

ICFP'11, September 19–21, 2011, Tokyo, Japan.

present paper, we report on this generalization, making the following contributions.

- We show that characteristic formulae for imperative programs can still be pretty printed in a way that closely resembles the source code they describe, in spite of the fact that their semantics now involves a memory store that is implicitly threaded throughout the execution of the program.
- In order to support local reasoning, we adapt characteristic formulae to handle specifications stated in the style of Separation Logic [39], and we introduce a predicate transformer for integrating the frame rule into characteristic formulae.
- We accompany the definition of characteristic formulae not only with a proof of soundness, but also with a proof of completeness. Completeness ensures that any correct specification can be established using characteristic formulae.
- We report on the verification of a nontrivial imperative algorithm, Dijkstra's shortest path algorithm. We also demonstrate the ability of CFML to reason about interactions between first-class functions and mutable state.

The content of this paper is organized in three main parts. Section 2 describes the key ideas involved in the construction, the pretty-printing and the manipulation of characteristic formulae for imperative programs. Section 3 gives details on the formalization of memory states, on the algorithm for generating characteristic formulae and on the soundness and completeness theorems. Section 4 contains a presentation of several examples that were specified and formalized using CFML. Due to space limitations, several aspects of CFML could only be summarized. All the details can be found in the author's PhD dissertation [8], and all the Coq proofs mentioned in this paper can be found online.[1]

2. Overview

2.1 Verification through characteristic formulae

The characteristic formula of a term t, written $[\![t]\!]$, relates a description of the input heap in which the term t is executed with a description of the output value and a description of the output heap produced by the execution of t. Characteristic formulae are hence closely related to Hoare triples [17], and, more precisely, to *total correctness* Hoare triples, which also account 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 $Q\,v$. Note that the post-condition Q is 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.

The characteristic formula $[\![t]\!]$ is a predicate 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. 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.

[1] `http://arthur.chargueraud.org/research/2011/cfml/`

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, and he is to provide information such as loop invariants during the interactive proof.

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, some facts need to be established in order to justify the correctness of the program. Those 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 let bindings, the frame rule and functions.

2.2 Characteristic formula of a let-binding

To evaluate a term of the form "$\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2$", one first evaluates the subterm t_1 and then computes the result of the evaluation of t_2, in which x denotes the result produced by t_1. To prove that the expression "$\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2$" admits H as pre-condition and Q as post-condition, one thus needs to find a valid post-condition Q' for t_1. This post-condition, when applied to the result x produced by t_1, describes the state of memory after the execution of t_1 and before the execution of t_2. So, $Q'\,x$ denotes the pre-condition for t_2. The corresponding Hoare-logic rule for reasoning on let-bindings is:

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

The characteristic formula for a let-binding is built as follows:

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

This formula closely resembles the corresponding Hoare-logic rule. The only real difference is that, 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 here. This existential quantification of specifications contrasts with traditional program verification approaches where intermediate specifications, including loop invariants, have to be included in the source code.

Next, we introduce a notation system for pretty-printing characteristic formulae. The aim is to make proof obligations easily readable and closely related to the source code. For let-bindings, the piece of notation defined is:

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

Hereafter, bold keywords correspond to notation for logical formulae, whereas plain keywords correspond to constructors from the

programming language syntax. The definition of the characteristic formula of a let-binding can now be reformulated as:

$$[\![\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2]\!] \quad \equiv \quad (\mathbf{let}\ x = [\![t_1]\!]\ \mathbf{in}\ [\![t_2]\!])$$

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.

Notation for characteristic formulae can be defined in a similar fashion 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. Hence, during the verification of a program, a proof-obligation appears to the user as a piece of source code followed with its pre-condition and its post-condition. 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 `xlet` applies to a goal of the form "$(\mathbf{let}\ x = \mathcal{F}_1\ \mathbf{in}\ \mathcal{F}_2)\ H\ Q$". It introduces a unification variable, call it Q', and produces two subgoals. The first one is $\mathcal{F}_1\ H\ Q'$. The second one is $\mathcal{F}_2\ (Q'\ x)\ Q$, under a context extended with a fresh variable named x. 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 `xlet`. More generally, CFML provides one such "x-tactic" for each language construction. As a result, one can 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 [36] refers to the ability to verify a piece of code by reasoning only about the memory cells that are involved in the execution of that 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 [39]. 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.

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

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.

To integrate the frame rule in characteristic formulae, we rely on a predicate called local. 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 $[\![t]\!]\ H_1\ Q_1$. Intuitively, the predicate local can be defined as follows.

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

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. Since we do not know where to insert applications of the predicate local, we may simply insert applications of this predicate at every node of characteristic formulae. For example, the previous definition for let-bindings gets updated as follows.

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

This aggressive strategy 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 specification of the empty heap.) We will later generalize the approach described here for handling the frame rule so as to also handle applications of the rule of consequence, which is used to strengthen pre-conditions and weaken post-conditions, and to enable the discarding of memory cells, for simulating garbage collection.

2.4 Translation of types

Higher-order logic can naturally be used to state properties about basic values such as purely-functional lists. Indeed, the list data structure defined in Coq perfectly matches the list data structure from Caml. However, particular care is required when specifying and reasoning about program functions. Indeed, programming language functions cannot be directly represented as logical functions, because of a mismatch between the two: program functions may be partial, whereas logical functions must always be total. To address this issue, we introduce a new data type, called Func, used to represent functions. To the user of characteristic formulae, the type Func is presented as an abstract data type. In the proof of soundness, however, a value of type Func is interpreted as the syntax of the source code of a function.

Another particularity of the reflection of program values into Coq values is the treatment of pointers. When reasoning through characteristic formulae, the type and the contents of memory cells are described explicitly through heap predicates, so there is no need for pointers to carry the type of the memory cell they point to. All pointers are therefore described in the logic through an abstract data type called Loc. In the proof of soundness, a value of type Loc is interpreted as a store location.

The translation of Caml types into Coq types is formalized through an operator, written $\langle\cdot\rangle$, that maps all arrow types to the type Func and maps all reference types to the type Loc. A Caml value of type τ is thus represented as a Coq value of type $\langle\tau\rangle$. For simplicity, program integers are idealized and are simply mapped to Coq values of type $\mathbb{Z}$. However, it would also be possible to map the type int to the Coq type $\mathsf{int64}$ for reasoning about overflows. The definition of the operator $\langle\cdot\rangle$ can be summarized as follows.

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

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, although memory safety is not obtained by weak-ML, it is guaranteed by the proofs of correctness established using a characteristic formula generated from a well-typed weak-ML program.

Although it is possible to generate characteristic formulae directly from ML programs, 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, however it is described in author's PhD dissertation [8].

2.5 Reasoning about functions

To specify the behavior of functions, we rely on a predicate, called App, which also appears 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 Heap $\to$ Prop, and the post-condition Q has type $B \to$ Hprop. So, the predicate App has type:

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

For example, the specification of the function incr, which increments the content of a memory cell containing an integer, takes the form of a theorem stated in terms of the predicate App:

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

Above, the heap predicate $(r \hookrightarrow n)$ describes the memory state expected by the function: it consists of a single memory cell located at address r and whose content is the value n. Similarly, the heap predicate $(r \hookrightarrow n+1)$ describes the memory state posterior to the function execution. The abstraction "λ_." is used to discard the unit value returned by the function incr.

By construction, a statement of the form "$\mathsf{App}\, f\, v\, H\, Q$" describes the behavior of an application. As we have just seen, App can be used to write 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. If a function f is defined as the abstraction "$\lambda x.\, t$", then, given a particular argument v, one can 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.

In what follows, we explain how to build characteristic formula for local functions and then for top-level function. For a local function definition, the characteristic formula is as follows:

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

For a top-level function definition of the form "$\mathsf{let\ rec}\, f = \lambda x.\, t$", CFML generates two Coq axioms. The first one has name f and type Func. This Coq variable f corresponds to the Caml function f. The second axiom describes the semantics of f, through the following statement: "$\forall x\, H\, Q.\ [\![t]\!]\, H\, Q \Rightarrow \mathsf{App}\, f\, x\, H\, Q$". Note that the soundness theorem proved for characteristic formulae ensures that adding this axiom does not introduce any logical inconsistency.

For example, consider the top-level function definition "$\mathsf{let}\, f = \lambda r.\, (\mathsf{incr}\, r\, ;\, \mathsf{incr}\, r)$", which expects a reference and increments its content twice. This function may be specified through a theorem whose statement is "$\forall r n.\ \mathsf{App}\, f\, r\, (r \hookrightarrow n)\, (\lambda\_.\ r \hookrightarrow n+2)$". To establish this theorem, the first step consists in applying the second axiom generated for the function f. The resulting proof obligation is "$(\textbf{app}\ \mathsf{incr}\, r\, \textbf{;}\ \textbf{app}\ \mathsf{incr}\ r)\, (r \hookrightarrow n)\, (\lambda\_.\ r \hookrightarrow n+2)$", 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.

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 [18].

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). This statement may also 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

This section of the paper explains in more details how characteristic formulae are constructed. It presents weak-ML types, the source language, the translation of Caml values into Coq values, and the predicates used to describe heaps. It then describes the algorithm used to generated characteristic formulae. Note that it is safe to read Section 4, which is concerned with examples, before this one.

3.1 From ML types to Weak-ML types and Coq types

In what follows, we describe the grammar of ML types and weak-ML types, and then formalize the translation from ML types to weak-ML types, and the translation from weak-ML types to Coq types. Hereafter, A denotes a type variable, C denotes the type constructor for an algebraic data type, τ denotes an ML type, and σ denotes a ML type scheme. Furthermore, the overbar notation denotes a list of items. The grammar of ML types is:

$$\tau \ := \ A \ \mid \ \mathsf{int} \ \mid \ C\, \overline{\tau} \ \mid \ \tau \to \tau \ \mid \ \mathsf{ref}\, \tau \ \mid \ \mu A.\tau$$
$$\sigma \ := \ \forall \overline{A}.\tau$$

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

$$
\begin{array}{lcl}
\langle A \rangle & \equiv & A \\
\langle \mathsf{int} \rangle & \equiv & \mathsf{int} \\
\langle C\,\overline{\tau} \rangle & \equiv & C\,\langle \overline{\tau} \rangle \\
\langle \tau_1 \to \tau_2 \rangle & \equiv & \mathsf{func} \\
\langle \mathsf{ref}\,\tau \rangle & \equiv & \mathsf{loc} \\
\langle \forall \overline{A}.\,\tau \rangle & \equiv & \forall \overline{B}.\,\langle \tau \rangle \quad \text{where } \overline{B} = \overline{A} \cap \mathsf{fv}(\langle \tau \rangle) \\
\langle \mu A.\tau \rangle & \equiv & \left| \begin{array}{ll} \langle \tau \rangle & \text{if } A \notin \langle \tau \rangle \\ \text{program rejected} & \text{otherwise} \end{array} \right.
\end{array}
$$

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

Weak-ML types are obtained from ML types by mapping all arrow types to a constant type called func and mapping all reference types to the constant type called loc. 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 translation of an ML type τ into the 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 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. Nevertheless, some 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 "$\mu A.(A \times \mathsf{int})$" does not have any counterpart in weak-ML, however the recursive ML type "$\mu A.(A \to B)$" gets mapped to the weak-ML type func. The verification approach described in the present paper therefore supports reasoning about functions with an equi-recursive type.

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. Note that the positivity requirement associated with Coq inductive types is not a problem here: since there is no arrow type in weak-ML, the translation from weak-ML types to Coq types never produces a negative occurrence of an inductive type in its own definition. In summary, the Coq translation of a weak-ML type T, written $\llbracket T \rrbracket$, is defined as follows.

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

3.2 Typed source language

Before generating characteristic formulae, programs first need to be put in an *administrative normal form*. Through this process, programs are arranged so that all intermediate results and all functions become bound by a let-definition. One notable exception is the application of simple total functions such as addition and subtraction. For example, the application "$f\,(v_1 + v_2)$" is considered to be in normal form although "$f\,(g\,v_1\,v_2)$" is not in normal form in general. The normalization process, which is similar to A-normalization [13], preserves the semantics and greatly simplifies formally reasoning about programs. Moreover, it is straightforward to implement. Similar transformations have appeared in previous work on program verification (e.g., [18, 38]). 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 typed, in the sense that all its subterms should be annotated with their weak-ML type. To formally define characteristic formulae, we therefore need to introduce the syntax of typed programs in normal forms. This syntax is formalized as follows, where $\hat{t}$ ranges over typed term and $\hat{v}$ ranges over typed values.

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

Note that locations and function closures do not exist in source programs, so they are not included in the above grammar. The letter n denotes an integer a memory location. The functions ref, get and set are used to allocate, read and write reference cells, respectively, and the function cmp enables comparison of two memory locations. The null pointer, written null, is a particular location that never gets allocated. Typed programs carry explicit information about generalized type variables, so a polymorphic function definition takes the form "$\mathsf{let\,rec}\ f = \Lambda\overline{A}.\lambda x.\hat{t}_1\ \mathsf{in}\ \hat{t}_2$" and 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. The syntax of typed programs also 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 later on (§3.7).

3.3 Reflection of values in the logic

Constructing characteristic formulae requires a translation of all the Caml values that appear in the program source code into the corresponding Coq values. 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 $\llbracket T \rrbracket$. The definition of $\lceil \hat{v} \rceil$ is shown below. Values on the left-hand side are well-typed weak-ML values whereas values on the right-hand side are (well-typed) Coq values.

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

Above, 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 application of x to the translations of each of the types from the list $\overline{T}$. A program data constructor D is mapped to the corresponding Coq inductive constructor, and if the constructor is polymorphic then its type arguments get translated into Coq 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 some 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. The Coq expert

might feel sceptical about the fact that the type variables A and B get assigned the kind Type. Since a weak-ML type variable is to be instantiated with a weak-ML type T, a Coq type variable occuring in a characteristic formula should presumably be instantiated only with a Coq type of the form $\lceil T \rceil$. Nevertheless, we have proved that it is not needed to consider the kind defined as the image of the operator $\lceil \cdot \rceil$, because it remains sound to assign the kind Type to the type variables quantified in characteristic formulae. The proof can be found in [8], Section 6.4.

3.4 Heap predicates

This section explains how heaps are represented, how operations on heaps are defined, and how heap predicates are built in the style of Separation Logic. Note that all the operations and predicates on heaps are completely formalized in Coq.

The semantics of a source program involves a memory store, which is a finite map from locations to program values. The Coq object that corresponds to a memory store is called a *heap*. 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 of a Coq type $\mathcal{T}$ and of a Coq value V of type $\mathcal{T}$. With this definition, the set of Coq values of type Heap is isomorphic to the set of well-typed memory stores.

Operations on heaps are defined in terms of operations on maps. The empty heap, written $\varnothing$, is a heap built on the empty map. Similarly, a singleton heap, written $l \rightarrow_{\mathcal{T}} V$, is a heap built on a singleton map binding a location l to a dependent pair made of a type $\mathcal{T}$ and a value V of type $\mathcal{T}$. Two heaps are said to be disjoint, written $h_1 \perp h_2$, when their underlying maps have disjoint domains. The union of two heaps, written $h_1 + h_2$, returns the union of the two underlying finite maps. We are only concerned with disjoint unions here, so it does not matter how the map union operator is defined for maps with overlapping domains.

Using those basic operations on heaps, one can define predicates for specifying heaps in the style of Separation Logic, as is done for example in Ynot [10]. Heap predicates are simply predicates over values of type Heap, so they have the type $\mathsf{Heap} \rightarrow \mathsf{Prop}$, abbreviated 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 $l \hookrightarrow_{\mathcal{T}} V$, which 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 defined as $\lambda h.\ \exists h_1\, h_2.\ h_1 \perp h_2 \ \wedge\ h = h_1 + h_2 \ \wedge\ H_1\, h_1 \ \wedge\ H_2\, h_2$.

In order to describe local invariants of data structures, propositions are lifted as heap predicates. More precisely, the predicate $[\mathcal{P}]$ holds of an empty heap if the proposition $\mathcal{P}$ is true. So, $[\mathcal{P}]$ is defined as $\lambda h.\ \mathcal{P} \wedge h = \varnothing$. In particular, the empty heap is characterized by the predicate $[\,]$, which is short for $[\mathsf{True}]$. Similarly, existential quantifiers are lifted: $\exists x.\, H$ holds of a heap h if there exists a value x such that H holds of that heap[2].

The present work ignores the disjunction construct $(H_1 \vee H_2)$. To reason on the content of the heap by case analysis, we instead rely on heap predicates of the form "$\mathsf{if}\ P\ \mathsf{then}\ H_1\ \mathsf{else}\ H_2$", which are defined using the builtin conditional construct from classical logic. The present work also does not make use of non-separating conjunction $(H_1 \wedge H_2)$. It therefore does not include the rule of conjunction, which can be found in a number of formalizations of Separation Logic. From a pratical perspective, we never felt the need for the conjunction rule. From a theoretical perspective, the conjunction rule is not needed for characteristic formulae to achieve completeness. (It is not yet known whether characteristic formulae would be able to accomodate the conjunction rule or not.)

Reasoning about heaps is generally conducted in terms of an entailment relation, written $H_1 \rhd H_2$, which asserts that any heap satisfying H_1 also satisfies H_2. It is defined as $\forall h.\, H_1\, h \Rightarrow H_2\, h$. Similarly, an entailment relation is provided for post-conditions. It is written $Q_1 \blacktriangleright Q_2$ and defined as $\forall x.\, Q_1\, x \rhd Q_2\, x$. A number of lemmas (not shown) allow reasoning about heap entailment without having to unfold the definition of this relation. Moreover, several tactics are provided to automate the application of these lemmas. As a result, apart from the setting up of the core definition and lemmas in the CFML library, the proofs never refer to objects of type Heap directly: program verification is carried out solely in terms of heap predicates of type Hprop (like done, e.g., in Ynot [10]).

Observe that the Separation Logic used here is not intuitionistic. In general, the entailment $H_1 * H_2 \rhd H_1$ is false. (It only holds when H_2 describes an empty heap.) With an intuitionistic Separation Logic, one may discard pieces of heap at any time during the reasoning on heap entailment. Here, garbage collection is instead modelled by having an explicit garbage heap mentioned in the definition of the predicate local, as described next.

3.5 Local predicates

In the introduction, we suggested how to define the predicate transformer "local" to account for applications of the frame rule. We now present the general definition of this predicate, a definition that also accounts for the rule of consequence and for the rule of garbage collection. Moreover, it supports the extraction of propositions and existentially-quantified variables from pre-conditions. We also introduce a predicate, called "islocal", that is useful for manipulating formulae of the form "$\mathsf{local}\, \mathcal{F}$".

The predicate local applies to a formula $\mathcal{F}$ with a type of the form $\mathsf{Hprop} \rightarrow (A \rightarrow \mathsf{Hprop}) \rightarrow \mathsf{Prop}$, for some type A. Its definition is:

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

where H describes the initial heap, H_1 corresponds to the part of the heap with which the formula $\mathcal{F}$ is concerned, H_2 corresponds to the part of the heap that is being framed out, H_3 corresponds to the part of the heap that gets discarded, Q describes the final result and final heap, and Q_1 is such that Q is equivalent to $Q_1 \star H_2$. (Recall that the latter is defined as $\lambda x.\ Q_1\, x * H_2$.) Note that the definition of the predicate local shows some similarities with the definition of the "STsep" monad from Hoare Type Theory [32], 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.

One can prove that the predicate local may be safely 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 "$\mathsf{local}\, (\mathsf{local}\, \mathcal{F})$". Other properties of local can be expressed in terms of a predicate called islocal, defined as:

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

This definition asserts that the predicate $\mathcal{F}$ is extensionally equivalent to "$\mathsf{local}\, \mathcal{F}$". In such a case, the formula $\mathcal{F}$ is called a *local* formula. Note that "$\mathsf{islocal}\, (\mathsf{local}\, \mathcal{F})$" is true for any $\mathcal{F}$.

Now, assuming that $\mathcal{F}$ is a local formula, all the reasoning rules shown in Figure 2 can be exploited. The interest of introducing the predicates islocal is that it conveniently allows us to apply any of the reasoning rules from Figure 2, an arbitrary number of times,

[2] The formal definition for existentials properly handles binders. It actually takes the form hexists J, where J is a predicate. Formally:
$\mathsf{hexists}\, (A : \mathsf{Type})\, (J : A \rightarrow \mathsf{Hprop}) \ \equiv\ \lambda(h : \mathsf{Heap}).\ \exists (x : A).\, J\, x\, h.$

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

Figure 2. Reasoning rules applicable to a *local* formula $\mathcal{F}$

$$
\begin{array}{l}
[\![\hat{v}]\!] \;\equiv \\
\quad \mathsf{local}\,(\lambda HQ.\ H \rhd Q\,\lceil\hat{v}\rceil) \\
[\![\hat{v}_1\,\hat{v}_2]\!] \;\equiv \\
\quad \mathsf{local}\,(\lambda HQ.\ \mathsf{App}\,\lceil\hat{v}_1\rceil\,\lceil\hat{v}_2\rceil\,H\,Q) \\
[\![\mathsf{let}\,x = \hat{t}_1\,\mathsf{in}\,\hat{t}_2]\!] \;\equiv \\
\quad \mathsf{local}\,(\lambda HQ.\ \exists Q'.\ [\![\hat{t}_1]\!]\,H\,Q' \wedge\ \forall x.\ [\![\hat{t}_2]\!]\,(Q'\,x)\,Q) \\
[\![\hat{t}_1\,;\,\hat{t}_2]\!] \;\equiv \\
\quad \mathsf{local}\,(\lambda HQ.\ \exists Q'.\ [\![\hat{t}_1]\!]\,H\,Q' \wedge\ [\![\hat{t}_2]\!]\,(Q'\,tt)\,Q) \\
[\![\mathsf{let\,rec}\,f = \Lambda\overline{A}.\lambda x.\hat{t}_1\,\mathsf{in}\,\hat{t}_2]\!] \;\equiv \\
\quad \mathsf{local}\,(\lambda HQ.\ \forall f.\ \mathcal{H} \Rightarrow [\![\hat{t}_2]\!]\,H\,Q) \\
\qquad \text{with } \mathcal{H} \;\equiv\; \forall\overline{A}xH'Q'.\ [\![\hat{t}_1]\!]\,H'\,Q' \;\Rightarrow\; \mathsf{App}\,f\,x\,H'\,Q' \\
[\![\mathsf{if}\,\hat{v}\,\mathsf{then}\,\hat{t}_1\,\mathsf{else}\,\hat{t}_2]\!] \;\equiv \\
\quad \mathsf{local}\,(\lambda HQ.\quad (\lceil\hat{v}\rceil = \mathsf{true} \;\Rightarrow\; [\![\hat{t}_1]\!]\,H\,Q) \\
\qquad\qquad \wedge\ (\lceil\hat{v}\rceil = \mathsf{false} \;\Rightarrow\; [\![\hat{t}_2]\!]\,H\,Q)) \\
[\![\mathsf{crash}]\!] \;\equiv \\
\quad \mathsf{local}\,(\lambda HQ.\ \mathsf{False}) \\
[\![\mathsf{let}\,x = \Lambda\overline{A}.\,\hat{v}\,\mathsf{in}\,\hat{t}]\!] \;\equiv \\
\quad \mathsf{local}\,(\lambda HQ.\ \forall x.\ x = \lambda\overline{A}.\lceil\hat{v}\rceil \;\Rightarrow\; [\![\hat{t}]\!]\,H\,Q)
\end{array}
$$

Figure 3. Generation of characteristic formulae

and in any order. Moreover, the predicate islocal plays a key role in the characteristic formulae of for-loops and while-loops (see §3.7).

3.6 Characteristic formula construction

We are now ready to describe the algorithm for constructing 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 $\mathsf{Hprop} \to (\lceil\!|T|\!\rceil \to \mathsf{Hprop}) \to \mathsf{Prop}$. Recall that Hprop is an abbreviation for $\mathsf{Heap} \to \mathsf{Prop}$. The rules for constructing characteristic formulae appear in Figure 3. Before describing each rule individually, two observations are worth making about the figure. First, every definition starts with an application of the predicate local. The presence of this predicate at every node of a characteristic formula enables us to apply any of the reasoning rules from Figure 2 at any point during the verification of a program. Second, all the program values get translated into Coq values. This is done through applications of the decoding operator, written $\lceil\hat{v}\rceil$.

The first rule from Figure 3 states that a value v admits a precondition H and a post-condition Q if the current heap, which is described by H, also satisfies the predicate $Q\,\lceil\hat{v}\rceil$. The characteristic formula of an application is obtained directly by applying the special predicate App. The treatment of let-bindings has already been explained in the introduction. The case of a sequence is a specialized version of that of let-bindings, where the result of the first term is always the unit value (written tt).

The treatment of functions has also already been explained, except for the treatment of polymorphism. 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 given the kind Type in Coq. Note that, in weak-ML, a polymorphic function admits the type func, just like any other function. So, the variable f admits in Coq the type Func.

To show that a conditional of the form "$\mathsf{if}\,v\,\mathsf{then}\,t_1\,\mathsf{else}\,t_2$" admits a given specification, one needs to prove that t_1 admits that specification when v is true and that t_2 admits that same specification when v is false. The definition of the characteristic formula of the instruction crash, which corresponds to a dead branch in the code, requires the programmer to prove that this point in the code can never be reached. This is equivalent to showing that the set of assumptions accumulated before reaching this point contains a logical inconsistency, i.e., that False is derivable.

The last definition from Figure 3 is slightly more technical. A polymorphic let-binding takes 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 $\forall\overline{A}.\lceil\!|T|\!\rceil$, and it 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 application "$x\,\lceil\!|\overline{U}|\!\rceil$" yields the Coq value that corresponds to the program value $\lceil\overline{A} \to \overline{U}\rceil\,\hat{v}$.

This completes the description of Figure 3. The characteristic formulae of loops are explained in the next section. The treatment of n-ary functions, mutually-recursive functions, assertions and pattern matching could not be described in this paper due to space limitations. This material can be found in the author's dissertation [8].

For each construction of the programming language, a custom Coq notation is defined for pretty-printing it in a way that resembles the source code. We have already seen how to pretty-print formulae for let-bindings. Additional examples concerning values, applications and function definitions are shown below.

$$
\begin{array}{l}
(\mathbf{ret}\ V) \;\equiv\; \mathsf{local}\,(\lambda HQ.\ H \rhd Q\,V) \\
(\mathbf{app}\ V_1\ V_2) \;\equiv\; \mathsf{local}\,(\lambda HQ.\ \mathsf{App}V_1V_2HQ) \\
(\mathbf{let\,rec}f = (\mathbf{fun}\,\overline{A}\,x := \mathcal{F}_1)\ \mathbf{in}\ \mathcal{F}_2) \;\equiv\; \mathsf{local}\,(\lambda HQ. \\
\quad \forall f.\ (\forall\overline{A}xH'Q'.\ \mathcal{F}_1\,H'\,Q' \;\Rightarrow\; \mathsf{App}\,f\,x\,H'\,Q') \;\Rightarrow\; \mathcal{F}_2\,H\,Q)
\end{array}
$$

Finally, consider the specification of the functions for manipulating references:

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

Above, the functions being specified have type Func, 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.

3.7 Characteristic formulae for loops

Since the source language already contains recursive functions, there is, from a theoretical perspective, no need do discuss the treatment of loops. That said, loops admit direct characteristic formulae whose use greatly shortens verification proof scripts in practice. 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 "$(r \hookrightarrow n)*(s \hookrightarrow m)$" and the post-condition "$(r \hookrightarrow 0) * (s \hookrightarrow m + n)$". 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 and increment functions are called, and the state after their execution is described as "$(r \hookrightarrow n-1) * (s \hookrightarrow m+1)$". At this point, we need to reason about the nested occurrence of t, that is, about the subsequent iterations of the loop. To that end, we invoke the induction hypothesis and derive the post-condition "$(r \hookrightarrow 0) * (s \hookrightarrow (m+1)+(n-1))$", which matches the required post-condition.

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, 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 the help of the notation introduced for pretty-printing characteristic formulae. The characteristic formula for while loops is therefore as follows. (The role of the hypothesis "$\mathsf{islocal}\,R$" is explained afterwards.)

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

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

To reflect the fact that the predicate R supports application of the frame rule as if it were a characteristic formula, the definition shown above provides the assumption that R is a local 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 this 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 the list trasversal 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 recently made by Tuerk [41].

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, here named i. The definition is:

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

3.8 Soundness and completeness

Characteristic formulae are both sound and complete. 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 a state h satisfying the pre-condition H, terminates and produces a value v in a final state h' such that the post-condition Q holds of v and h'. The semantics judgment involved here is written $\hat{t}_{/h} \Downarrow \hat{v}_{/h'}$. The formal statement shown below also takes into account the fact the final heap may contain some garbage values, which are gathered in a sub-heap called h''.

Theorem 3.1 (Soundness) *Let $\hat{t}$ be a well-typed, closed weak-ML term. Let H and Q be a pre- and a post-condition, and h be a heap.*

$$[\![\hat{t}]\!]\,H\,Q \;\wedge\; H\,h \;\Rightarrow\; \exists \hat{v}\,h'\,h''.\;\hat{t}_{/h} \Downarrow \hat{v}_{/(h'+h'')} \;\wedge\; Q\,\lceil \hat{v} \rceil\,h'$$

Above, H has type "$\mathsf{Heap} \to \mathsf{Prop}$" and Q has type "$[\![T]\!] \to \mathsf{Heap} \to \mathsf{Prop}$", where T is the type of $\hat{t}$.

The completeness theorem asserts that, reciprocally, if a program admits a given specification, then it is possible to prove that the characteristic formula of this program holds of that specification. This completeness statement is, of course, relative to the expressiveness power of the logic of Coq. More precisely, the statement of completeness states the following: if one is 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 it is possible to establish in Coq that the characteristic formula of this program holds of the given post-condition.

Due to space limitations, the present paper does not include the general statement of the completeness theorem, which involves the notion of most-general specification and that of typed reduction, but only a specialized version for 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 3.2 (Completeness —particular case) *Let t be a closed ML term, and let $\hat{t}$ denote the corresponding weak-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 $t_{/\emptyset} \Downarrow n_{/h}$ is interpreted as the statement of a fact provable in Coq. More precisely, this hypothesis asserts the existence of a Coq proof term witnessing the fact that the configuration $t_{/\emptyset}$ is related to the configuration $n_{/h}$ by the inductively-defined evaluation judgment ($\Downarrow$).

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 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. Examples

This section describes four examples. The first one is Dijsktra's shortest path algorithm. It illustrates how CFML supports the reasoning about modular code involving complex invariants. The other

examples focus on the treatment of imperative first-class functions, covering a counter function with an abstract local state, Reynold's CPS-append function, and an iterator on imperative lists.

Conducting proofs using CFML involves two additional ingredients that have not yet been described. The first one is the predicate App_n, which generalizes the predicate App to n-ary applications. For example, "$\mathsf{App}_2\ f\ x\ y\ H\ Q$" asserts that the application of f to x and y admits H and Q as pre- and post-conditions. The predicate App_1 is the same as App, and the predicates App_n can be defined in terms of App_1.

The second key ingredient is the notion of a representation predicate. A heap predicate of the form $v \rightsquigarrow T\,V$ is used to relate the mutable data structure found at location v with the mathematical value V that it represents. Here, T is a representation predicate: it characterizes the relationship between v, V and the piece of memory state spanned by the data structure under consideration. In fact, $v \rightsquigarrow T\,V$ is simply defined as $T\,V\,v$, where T can be any predicate of type $A \to B \to \mathsf{Hprop}$. This section contains examples showing how to use and how to define representation predicates.

4.1 Dijkstra's shortest path

In this first example, describe the specification and verification of a particular implementation of Dijkstra's algorithm. This implementation 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 for which the best distance is already known; b, an array of distances used to store the best know distance for every node (distances may be infinite); and q, a priority queue for efficiently identifying the next nodes to be visited.

The Caml source code is 20 lines long, and 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, whose argument corresponds to a priority queue module. Similarly, the verification script is implemented as a Coq functor. This functor expects two arguments: a module representing the implementation of the priority queue, and a module representing the proofs of correctness of that queue implementation. This strategy allows us to achieve modular verification of modular code.

The specification of the function dijkstra is as follows:

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

It 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 weights, 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. Moreover, the above specification asserts that the structure of the graph is not modified by the execution of the function.

The representation predicate GraphAdjList is used to relate a mathematical graph with its representation as an array of lists of pairs. It is defined as:

$$\begin{array}{l}\mathsf{GraphAdjList}\,G\,g \ \equiv\ \exists N.\ (g \rightsquigarrow \mathsf{Array}\,N) * \\ \quad [\forall x.\ x \in \mathsf{nodes}\,G \Leftrightarrow x \in \mathsf{dom}\,N] * \\ \quad [\forall x \in \mathsf{nodes}.\,\forall y w.\ (x,y,w) \in \mathsf{edges}\,G \Leftrightarrow \mathsf{mem}(y,w)\,N[x]]\end{array}$$

Above, g denotes a value of type Loc, G denotes a mathematical graph whose nodes are indexed by integers and whose edges have integer weight, and N is a finite map from integers to lists of pairs of integers. The definition asserts that x is an index in N if and only if it is the index of a node in G, and 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 invariant of the main loop of Dijkstra's algorithm, written "$\mathsf{hinv}\,V\,B\,Q$" describes the state of the data structures in terms of three data structures: V is a finite map describing the array v, B is a finite map describing the array b, and Q is a multiset of pairs describing the priority queue q. Several logical invariants enforce constraints ocharacteristic formulae.n the content of V, B and Q. Those 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.

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

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 one theorem, whose statement is 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, and therefore also follows the structure of the source code.

Figure 4 show the beginning of the proof script for this verification theorem. 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, general-purpose Coq tactics (all those whose name does not start with the letter "x") are typically used to name variables, unfold invariants, and discharge simple side-conditions. Third, the proof script contains invocations of the mathematical theorems mentioned earlier on. 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, this 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.

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. Those names are provided explicitly in the proof script. Providing names is not mandatory, however it generally helps to increase readability and robustness. The proof obligation appears below the dashed line. It consists of a characteristic formula being applied to a pre-condition and to a post-condition. Note that, in Coq, characteristic formula are pretty-printed 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.
```

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

```
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)
```

Figure 5. Example of a proof obligation

4.2 Counter function

This example illustrates the treatment of functions 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 constructs a new counter function. It allocates a fresh reference r with initial contents 0, and builds a function whose body increments r and returns its contents.

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

To specify the function create in an abstract manner, we use a representation predicate, called Cntr. The heap predicate "$f \rightsquigarrow \mathsf{Cntr}\, n$" 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 "int $\to$ Hprop", as shown below:

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

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.

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

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

This specification is sufficient for reasoning about all the calls to a counter function produced by the function create. That said, we can go even further in terms of abstraction. Instead of forcing the client of the function create to manipulate the definition of Cntr, we can make the definition of the predicate Cntr completely abstract and instead provide a direct lemma for reasoning about calls to counter functions. This lemma takes the following form:

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

This example illustrates how the abstract local state of a function can be entirely packed into a representation predicate.

4.3 Continuations

The CPS-append function has been proposed as a verification challenge by Reynolds [39], 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. In the end, the function 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 those 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 is:

```
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))
```

Its specification is as follows, where k has type Func, x and y have type "list A", and $+\!\!+$ denotes the concatenation of two Coq lists:

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

Slightly more challenging is the verification of the imperative counterpart of the CPS-append function. 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 is:

$$\forall A x y k L M H Q.\; (\forall z.\, \mathsf{App}_1\, k\, z\, (H * (z \rightsquigarrow \mathsf{Mlist}\, (L{+\!\!+}M)))\, Q) \;\Rightarrow\; \mathsf{App}_3\, \mathsf{cpsapp'}\, x\, y\, k\, (H * (x \rightsquigarrow \mathsf{Mlist}\, L) * (y \rightsquigarrow \mathsf{Mlist}\, M))\, Q$$

Above, the pre-condition asserts that the locations x and y (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.

4.4 Imperative list iterator

This last example requires a generalized version of the representation predicate for lists. So far, we have used heap predicates of the form $m \rightsquigarrow \mathsf{Mlist}\, L$. This works well when the values in the list are of some base type, however in general the values stored in the list

[3] Thielecke [40] suggested 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.

need to be described using their own representation predicate, call it T. To that end, we use a more general parametric representation predicate, written $\mathsf{Mlistof}\,T$. (The predicate Mlist used so far can be obtained as the application of $\mathsf{Mlistof}$ to the identity representation predicate, which is defined as "$\lambda X.\,\lambda x.\,[x = X]$".) For example, we will later use the heap predicate "$m \rightsquigarrow \mathsf{Mlistof\ Cntr}\,L$" to describe a mutable list that starts at location m and contains a list of counter functions whose internal states are described by the integer values from the Coq list L.

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. The execution of "$\mathsf{iter}\,f\,m$" results in the function f being applied to all the values stored in the list whose head is located as address m. This execution may result in two effects. First, it may modify the values stored in the list. Second, it may affect the state of other mutable data structures. Thus, if the initial state is described as $H * (m \rightsquigarrow \mathsf{Mlistof}\,T\,L)$, then the final state generally takes the form $H' * (m \rightsquigarrow \mathsf{Mlistof}\,T\,L')$, 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 $H * (m \rightsquigarrow \mathsf{Mlistof}\,T\,L)$ and the post-condition $\lambda\_.\ H' * (m \rightsquigarrow \mathsf{Mlistof}\,T\,L')$.

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 one concerns the case where the list is not empty. In this case, a call to f is first performed and then a recursive call to the function iter is made. The initial state of the list is then of the form $X :: L$ and the final state is 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:

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

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 $H * (m \rightsquigarrow \mathsf{Mlistof}\,T\,L)$ and the post-condition $\lambda\_.\ H' * (m \rightsquigarrow \mathsf{Mlistof}\,T\,L')$, it suffices to prove the proposition $I\,L\,L'\,H\,H'$, where I is an abstract predicate for which only the assumptions $\mathcal{H}_1$ and $\mathcal{H}_2$ are provided. This result is captured by the specification of iter shown next:

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

To check the usability of this specification, we describe an example, which involves a list m of distinct counter functions (as defined in §4.2). The idea is to make a call to each of those counters. The results of those 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 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. Thus, steps is specified as:

$$\begin{array}{l}
\forall m L.\ \mathsf{App}_1\,\mathsf{steps}\,m\,(m \rightsquigarrow \mathsf{Mlistof\ Cntr}\,L) \\
\qquad\qquad (\lambda_.\ m \rightsquigarrow \mathsf{Mlistof\ Cntr}\,(\mathsf{map}\,(+1)\,L))
\end{array}$$

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

5. Related work

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 [17] is probably the most well-known program logic. Separation Logic [39] is an extension of Hoare logic that supports local reasoning. A number of verification tools have been built upon ideas from Separation Logic, for example Smallfoot [5]. Separation Logic frequently been exploited inside standard interactive proof assistants (e.g., [1, 10, 26, 27, 30]), including the present paper. Dynamic Logic [15] is another program logic. In this modal logic, the formula "$H_1 \to \langle t \rangle\, H_2$" 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 of the KeY system [4], which targets the verification of Java programs. One problem with Dynamic Logics is that they depart from standard mathematical logic, precluding the use of a standard proof assistant.

The aforementioned logics usually do not support reasoning about higher-order functions. A program logic supporting them has been 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 the predicate App. Another specificity of the specification language is that its values are the values of the programming language, including non-terminating functions. This use of such a nonstandard specification language prevented Honda *et al* from building a practical verification tool on top of an existing theorem prover. In contrast, the characteristic formulae that we generate are expressed in terms of a standard higher-order logic predicates.

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

A few tools support higher-order logic. One notable example is the tool Why [12]. When the proof obligations produced by Why cannot be verified automatically by at an SMT solver, they can 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 [20], building upon ideas developed for the tool Pangolin [38]. Another tool that supports higher-order logic is Jahob [42], 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. When trying to verify complex programs, the central difficulty is to come up with the correct invariants, a process that usually requires a great 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 the VCG tool, wait for the SMT solvers to try and discharge the proof obligations, and then read the remaining obligations. On the contrary, with characteristic formulae, the user works in an interactive setting that provides nearly-instantaneous feedback on changes to the invariants.

Shallow embeddings The shallow embedding approach to program verification aims at relating a source program to a corresponding logical definition. The relationship can take three forms.

First, one can write a logical definition and use an extraction mechanism (e.g., [25]) to translate the code into a conventional programming language. For example, Leroy's certified C compiler [23] is developed in this way. Also based on extraction is the tool Ynot [10], which implements Hoare Type Theory (HTT) [33], 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* [34] 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 correponds to a partial-correctness specification with pre-condition P and post-condition Q. Verification proofs take the form of Coq typing derivations for the source code. So, program verification is done at the same time as type-checking the source code. This is a significant difference with characteristic formulae, which allow verifying programs after they have been written, without requiring the source code to be modified in any way. Moreover, characteristic formulae 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 handy features such as alias-patterns and when-clauses would be a real challenge for Ynot. (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 [10] relies on a hack for simulating the Implicit Calculus of Constructions [3], in which computationally-irrelevant value are tagged explicitly. A danger of this approach is that forgetting to tag a variable as auxiliary does not produce any warning yet results in the extracted code being inefficient. Other implementation of HTT have taken a different approach by relying on post-conditions that may also refer not only to the output heap but also to the input heap [33, 34]. The use of such *binary post-conditions* makes it possible to eliminate auxiliary variables by duplicating the pre-condition inside the post-condition. Typically, in informal notation, "$\forall x.\ \mathsf{STsep}\ P\ Q$" gets encoded as "$\mathsf{STsep}\ (\exists x.P)\ (\forall x.\ P\ \Rightarrow\ Q)$". HTT [34] then provides tactics to try and avoid the duplication of proof obligations. However, duplication typically remains visible in specifications, which is problematic. Indeed, 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 consists in decompiling a piece of conventional source code into a set of logical definitions. This approach is used in the LOOP compiler [19] and also in Myreen and Gordon's work [31]. The LOOP compiler takes Java programs and compiles them into PVS definitions. The 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 [19]. 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 [31]. 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. Importantly, the translation into HOL4 is possible only 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 seems hard 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 in 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 [22]. Compared with Myreen and Gordon's work [29, 31], the main difference is that the low-level code is not decompiled automatically but instead decompiled by hand, and that 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 bigger 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 [21].

In summary, all approaches based on 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.

Deep embeddings A deep embedding consists of describing 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 [28] 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 the frameworks such as XCAP [35] 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. This work then 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 explicit representation of syntax, which is associated with many technical difficulties (including the represen-

tation of binders). Moreover, when moving to characteristic formulae, specifications can be greatly simplified because program values such as tuples and functional lists become directly represented with their logical counterpart.

6. Conclusion

In this paper, we have explained how to build characteristic formulae for imperative programs, and we have shown how to use those formulae in practice to formally verify programs involving nontrivial interactions between first-class functions and mutable state.

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

[11] The Coq Development Team. *The Coq Proof Assistant Reference Manual, Version 8.2*, 2009.

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

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

[14] 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.

[15] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. The MIT Press, Cambridge, Massachusetts, 2000.

[16] Matthew Hennessy and Robin Milner. On observing nondeterminism and concurrency. In *ICALP*, volume 85 of *LNCS*, pages 299–309. Springer-Verlag, 1980.

[17] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, 1969.

[18] 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.

[19] Bart Jacobs and Erik Poll. Java program verification at nijmegen: Developments and perspective. In *ISSS*, volume 3233 of *LNCS*, pages 134–153. Springer, 2003.

[20] 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.

[21] Gerwin Klein, Philip Derrin, and Kevin Elphinstone. Experience report: seL4: formally verifying a high-performance microkernel. In *ICFP*, pages 91–96. ACM, 2009.

[22] 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.

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

[24] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system*, 2005.

[25] Pierre Letouzey. *Programmation fonctionnelle certifiée – l'extraction de programmes dans l'assistant Coq*. PhD thesis, Université Paris 11, 2004.

[26] Nicolas Marti, Reynald Affeldt, and Akinori Yonezawa. Towards formal verification of memory properties using separation logic, 2005.

[27] Andrew McCreight. Practical tactics for separation logic. In *TPHOLs*, volume 5674 of *LNCS*, pages 343–358. Springer, 2009.

[28] Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. *Information and Computation*, 199(1–2), 2005.

[29] Magnus O. Myreen. *Formal Verification of Machine-Code Programs*. PhD thesis, University of Cambridge, 2008.

[30] Magnus O. Myreen. Separation logic adapted for proofs by rewriting. In *Interactive Theorem Proving (ITP)*, volume 6172 of *LNCS*, pages 485–489. Springer, 2010.

[31] 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.

[32] Aleksandar Nanevski and Greg Morrisett. Dependent type theory of stateful higher-order functions. Technical Report TR-24-05, Harvard University, 2005.

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

[34] Aleksandar Nanevski, Viktor Vafeiadis, and Josh Berdine. Structuring the verification of heap-manipulating programs. In *POPL*, pages 261–274. ACM, 2010.

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

[36] 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.

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

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

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

[40] Hayo Thielecke. Frame rules from answer types for code pointers. In *POPL*, pages 309–319, 2006.

[41] Thomas Tuerk. Local reasoning about while-loops. In *VSTTE LNCS*, 2010.

[42] Karen Zee, Viktor Kuncak, and Martin C. Rinard. An integrated proof language for imperative programs. In *PLDI*, pages 338–351. ACM, 2009.