UNIVERSITÉ PARIS.DIDEROT (Paris 7)
ÉCOLE DOCTORALE : Sciences Mathématiques de Paris Centre

Ph.D in Computer Science

Characteristic Formulae for Mechanized Program Verification

Arthur Charguéraud

Advisor: François POTTIER

Defended on December 16, 2010

Jury

Claude	Marché	Reviewer
Greg	Morrisett	Reviewer
Roberto	di Cosmo	Examiner
Yves	Bertot	Examiner
Matthew	Parkinson	Examiner
François	Pottier	Advisor

Abstract

This dissertation describes a new approach to program verification, based on characteristic formulae. The characteristic formula of a program is a higher-order logic formula that describes the behavior of that program, in the sense that it is sound and complete with respect to the semantics. This formula can be exploited in an interactive theorem prover to establish that the program satisfies a specification expressed in the style of Separation Logic, with respect to total correctness.

The characteristic formula of a program is automatically generated from its source code alone. In particular, there is no need to annotate the source code with specifications or loop invariants, as such information can be given in interactive proof scripts. One key feature of characteristic formulae is that they are of linear size and that they can be pretty-printed in a way that closely resemble the source code they describe, even though they do not refer to the syntax of the programming language.

Characteristic formulae serve as a basis for a tool, called CFML, that supports the verification of Caml programs using the Coq proof assistant. CFML has been employed to verify about half of the content of Okasaki's book on purely functional data structures, and to verify several imperative data structures such as mutable lists, sparse arrays and union-find. CFML also supports reasoning on higher-order imperative functions, such as functions in CPS form and higher-order iterators.

Contents

Chapter 1

Introduction

The starting point of this dissertation is the notion of correctness of a program. A program is said to be correct if it behaves according to its specification. There are applications for which it is desirable to establish program correctness with a very high degree of confidence. For large-scale programs, only mechanized proofs, i.e., proofs that are verified by a machine, can achieve this goal. While several decades of research on this topic have brought many useful ingredients for building verification systems, no tool has yet succeeded in making program verification a routine exercise. The matter of this thesis is the development of a new approach to mechanized program verification. In this introduction, I explain where my approach is located in the design space and give a high-level overview of the ingredients involved in my work.

1.1 Approaches to program verification

Motivation Computer programs tend to become involved in a growing number of devices. Moreover, the complexity of those programs keeps increasing. Nowadays, even a cell phone typically relies on more than ten million lines of code. One major concern is whether programs behave as they are intended to. Writing a program that appears to work is not so difficult. Producing a fully-correct program has shown to be astonishingly hard. It suffices to get one single character wrong among millions of lines of code to end up with a buggy program. Worse, a bug may remain undetected for a long time before the particular conditions in which the bug shows up are gathered. If you are playing a game on your cell phone and a bug causes the game to crash, or even causes the entire cell phone to reboot, it will probably be upsetting but it will not have dramatic consequences. However, think of the consequences of a bug occurring in the cruise control of an airplane, a bug in the program running the stock exchange, or a bug in the control system of a nuclear plant.

Testing can be used to expose bugs, and a coverage analysis can even be used to

check that all the lines from the source code of a program have been tested at least once. Yet, although testing might expose many bugs, there is no guarantee that it will expose all the bugs. Static analysis is another approach, which helps detecting all the bugs of a certain form. For example, type-checking ensures that no absurd operation is being performed, like reading a value in memory at an address that has never been allocated by the program. Static analysis has also been successfully applied to array bound checking, and in particular to the detection of buffer overflow vulnerabilities (e.g., [84]). Such static analyses can be almost entirely automated and may produce a reasonably-small number of false positives.

Program verification Neither testing nor static analysis can ensure the total absence of errors. More advanced tools are needed to prove that programs always behave as intended. In the context of program verification, a program is said to be correct if it satisfies its specification. The specification of a program is a description of what a program is intended to compute, regardless of how it computes it. The general problem of computing whether a given program satisfies a given specification is undecidable. Thus, a general-purpose verification tool must include a way for the programmer to guide the verification process.

In traditional approaches based on a Verification Condition Generator (VCG), the transfer of information from the user to the verification system takes the form of invariants that annotate the source code. Thus, in addition to annotating every procedure with a precise description of what properties the procedure can expect of its input data and of what properties it guarantees about its output data, the user also needs to include intermediate specifications such as loop invariants. The VCG tool then extracts a set of proof obligations from such an annotated program. If all those proof obligations can be proved correct, then the program is guaranteed to satisfy its specification.

Another approach to transferring information from the user to the verification tool consists in using an interactive theorem prover. An interactive theorem prover, also called a proof assistant, allows one to state a theorem, for instance a mathematical result from group theory, and to prove it interactively. In an interactive proof, the user writes a sequence of tactics and statements for describing what reasoning steps are involved in the proofs, while the proof assistant verifies in real-time that each of those steps is legitimate. There is thus no way to fool the theorem prover: if the proof of a theorem is accepted by the system, then the theorem is certainly true (assuming, of course, that the proof system itself is sound and correctly implemented). Interactive theorem provers have been significantly improved in the last decade, and they have been successfully applied to formalize large bodies of mathematical theories.

The statement “this program admits that specification” can be viewed as a theorem. So, one could naturally attempt to prove statements of this form using a proof assistant. This high-level picture may be quite appealing, yet it is not so immediate to implement. On the one hand, the source code of a program is expressed in some

programming language. On the other hand, the reasoning about the behavior of that program is carried out in terms of a mathematical logic. The challenge is to find a way of building a bridge between a programming language and a mathematical logic. For example, one of the main issue is the treatment of mutable state. In a program, a variable x may be assigned to the value 5 at one point, and may later updated to the value 7. In mathematics, on the contrary, when x has the value 5 at one point, it has the value 5 forever. Devising a way to describe program behaviors in terms of a mathematical logic is a central concern of my thesis. Several approaches have been investigated in the past. I shall recall them and explain how my approach differs.

Interactive verification The deep embedding approach involves a direct axiomatization of the syntax and of the semantics of the programming language in the logic of the theorem prover. Through this construction, programs can be manipulated in the logic just as any other data type. In theory, this natural approach allows proving any correct specification that one may wish to establish about a given program. In practice, however, the approach is associated with a fairly high overhead, due to the clumsiness of explicit manipulation of program syntax. Nonetheless, this approach has enabled the verification of low-level assembly routines involving a small number of instructions that perform nontrivial tasks [65, 25].

Rather than relying on a representation of programs as a first-order data type, one may exploit the fact that the logic of a theorem prover is actually a sort of programming language itself. For example, the logic of Coq [18] includes functions, algebraic data structures, case analysis, and so on. The idea of the shallow embedding approach is to relate programs from the programming language with programs from the logic, despite the fact that the two kinds of programs are not entirely equivalent.

浅度嵌入

There are basically three ways to build on this idea. A first technique consists in writing the code in the logic and using an extraction mechanism (e.g., [49]) to translate the code into a conventional programming language. For example, Leroy's certified C compiler [47] is developed in this way. The second technique works in the other direction: a piece of conventional source code gets decompiled into a set of logical definitions. Myreen [58] has employed this technique to verify a machine-code implementation of a LISP interpreter. Finally, with the third approach, one writes the program twice: once in a deep embedding of the programming language, and once directly in the logic. A proof can then be conducted to establish that the two programs match. Although this approach requires a bigger investment than the former two, it also provides a lot of flexibility. This third approach has been employed in the verification of a microkernel, as part of the Sel4 project [46].

All approaches based on shallow embedding share one big difficulty: the need to overcome the discrepancies between the programming language and the logical language, in particular with respect to the treatment of partial functions and of mutable state. One of the most developed techniques for handling those features in a shallow embedding has been developed in the Ynot project [17]. Ynot, which is

HTT：Hoare Type Theory

based on Hoare Type Theory (HTT) [63], relies on a dependently-typed monad for describing impure computations.

The approach developed in this thesis is quite different: given a conventional source code, I generate a logical proposition that describes the behavior of that code. In other words, I generate a logical formula that, when applied to a specification, yields a sufficient condition for establishing that the source code satisfies that specification. This sufficient condition can then be proved interactively. By not representing explicitly program syntax, I avoid the technical difficulties related to the deep embedding approach. By not relying on logical functions to represent program functions, I avoid the difficulties faced by the shallow embedding approach.

通过不显式地表达程序语法，避免深度嵌入的困难
通过不依赖逻辑来表示程序，避免浅度嵌入的困难

特征公式简介

Characteristic formulae The formula that I generate for a given program is a sound and almost-complete description of the behavior of that program, hence it is called a characteristic formula. Remark: a characteristic formula is only "almost-complete" because it does not reveal the exact addresses of allocated memory cells and it does not allow specifying the exact source code of function closures. Instead, functions have to be specified extensionally. The notion of characteristic formulae dates back to the 80's and originates in work on process calculi, a model for parallel computations. There, characteristic formulae are propositions, expressed in a temporal logic, that are generated from the syntactic definition of a process. The fundamental result about those formulae is that two processes are behaviorally equivalent if and only if their characteristic formulae are logically equivalent [70, 57]. Hence, characteristic formulae can be employed to prove the equivalence or the disequivalence of two given processes.

仅几乎完全的原因
(只能用存在表示)：
1.不能确切地表示准确的地址
2.不允许带有函数闭包确切的源代码

More recently, Honda, Berger and Yoshida [40] adapted characteristic formulae from process logic to program logic, targetting PCF, the "Programming language for Computable Functions". PCF can be seen as a simplified version of languages of the ML family, which includes languages such as Caml and SML. Honda *et al* give an algorithm that constructs the "total characteristic assertion pair" of a given PCF program, that is, a pair made of the weakest pre-condition and of the strongest post-condition of the program. (Note that the PCF program is not annotated with any specification nor any invariant, so the algorithm involved here is not the same as a weakest pre-condition calculus.) This notion of most-general specification of a program is actually much older, and originates in work from the 70's on the completeness of Hoare logic [31]. The interest of Honda *et al*'s work is that it shows that the most general specification of a program can be expressed without referring to the programming language syntax.

Honda *et al* also suggested that characteristic formulae could be used to prove that a program satisfies a given specification, by establishing that the most-general specification entails the specification being targeted. This entailment relation can be proved entirely at the logical level, so the program verification process can be conducted without the burden of referring to program syntax. Yet, this appealing idea suffered from one major problem. The specification language in which total

characteristic assertion pairs are expressed is an ad-hoc logic, where variables denote PCF values (including non-terminating functions) and where equality is interpreted as observational equality. As it was not immediate to encode this ad-hoc logic into a standard logic, and since the construction of a theorem prover dedicated to that new logic would have required an tremendous effort, Honda *et al*'s work remained theoretical and did not result in an effective program verification tool.

通过构建逻辑公式，
来捕获能够进行深度嵌入的推理，
并且不需要在任何时候以程序语法表示。

I have re-discovered characteristic formulae while looking for a way to enhance the deep embedding approach, in which program syntax is explicitly represented in the theorem prover. I observed that it was possible to build logical formulae capturing the reasoning that can be done through a deep embedding, yet without exposing program syntax at any time. In a sense, my approach may be viewed as building an abstract layer on top of a deep embedding, hiding the technical details while retaining its benefits. Contrary to Honda, Berger and Yoshida's work, the characteristic formulae that I build are expressed in terms of standard higher-order logic. I have therefore been able to build a practical program verification tool based on characteristic formulae.

1.2 Overview

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 of the output heap produced by the execution of t. Characteristic formulae are closely related to Hoare triples [29, 36]. 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 value and the output heap. 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.

这里说的是
分离逻辑中的Hoare三元组

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 $\Rightarrow$, which 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.

In the rest of this section, I present the key ideas involved in the construction of characteristic formulae, focusing on the treatment of let bindings, of function applications, and of function definitions. I also explain how to handle the frame rule, which enables local reasoning.

Let-bindings 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 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 appears next.

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

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

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

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 in my work. This contrasts with traditional program verification approaches where intermediate specifications, including loop invariants, have to be included in the source code.

In order to make proof obligations more readable, I introduce a system of notation for characteristic formulae. For example, for let-bindings, I define:

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

Note that bold keywords correspond to notation for logical formulae, whereas plain keywords correspond to constructors from the programming language syntax. The generation of characteristic formulae then boils down to a re-interpretation of the keywords from the programming language.

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

It follows that characteristic formulae may be pretty-printed exactly like source code. Hence, the statement asserting that a term t admits a pre-condition H and a post-condition Q, which takes the form "$[\![t]\!]\, H\, Q$", appears to the user as the source code of t 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 proof of correctness of the term t.

Frame rule "Local reasoning" [67] refers to the ability of verifying a piece of code through a piece of reasoning concerned only with 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 [77]. 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 described by $H_1' * H_2$, where the star symbol, called separating conjunction, captures a disjoint union of two pieces of heap.

For example, consider the application of the function incr, which increments the contents of the memory cell, to a location l. If $(l \hookrightarrow n)$ describes a singleton heap that binds the location l to the integer n, then an application of the function incr expects a heap of the form $(l \hookrightarrow n)$ and produces the heap $(l \hookrightarrow n+1)$. By the frame rule, one can deduce that the application of the function incr to l also takes a heap of the form $(l \hookrightarrow n) * (l' \hookrightarrow n')$ towards a heap of the form $(l \hookrightarrow n+1) * (l' \hookrightarrow n')$. Here, the separating conjunction asserts that l' is a location distinct from l. The use of the separating conjunction gives us for free the property that the cell at location l' is not modified when the contents of the cell at location l is incremented.

The frame rule can be formulated on Hoare triples as follows.

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

where 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, I rely on a predicate called frame. This predicate is defined in such a way that, to prove the proposition "$\mathsf{frame}\, [\![t]\!]\, H\, Q$", it suffices to find a decomposition of H as $H_1 * H_2$, a decomposition of Q as $Q_1 \star H_2$, and to prove $[\![t]\!]\, H_1\, Q_1$. The formal definition of frame is thus as follows.

$$\mathsf{frame}\, \mathcal{F} \quad \equiv \quad \lambda H Q.\ \exists H_1\, H_2\, Q_1.\ \left\{ \begin{array}{l} H = H_1 * H_2 \\ \mathcal{F}\, H_1\, Q_1 \\ Q = Q_1 \star H_2 \end{array} \right.$$

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, I want to generate characteristic formulae in a systematic manner form the syntax of the source code. Since I do not know where to insert applications of the predicate frame, I simply insert applications of frame at every node of a characteristic formula. For example, I update the previous definition for let-bindings to:

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

This aggressive strategy allows applying the frame rule at any time in the reasoning. If there is no need to apply the frame rule, then the frame predicate may be simply ignored. Indeed, given a formula $\mathcal{F}$, the proposition "$\mathcal{F}\, H\, Q$" is always a sufficient

condition for proving "$\mathsf{frame}\,\mathcal{F}\,H\,Q$". It suffices to instantiate H_1 as H, Q_1 as Q and H_2 as a specification of the empty heap.

The approach that I have described here to handle the frame rule is in fact generalized so as to also handle applications of the rule of consequence, for strengthening pre-conditions and weakening post-conditions, and to allow discarding memory cells, for simulating garbage collection.

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 can be defined in the logic in a way that perfectly matches the list data structure from the programming language. However, particular care is required for specifying and reasoning on program functions. Indeed, programming language functions cannot be directly represented as logical functions, because of a mismatch between the two: program functions may diverge or crash whereas logical functions must always terminate. To address this issue, I introduce a new data type, called Func, used to represent functions. The type Func is presented as an abstract data type to the user of characteristic formulae. In the proof of soundness, a value of type Func is interpreted as the syntax of the source code of a function.

Another particularity of the reflection of Caml 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 by 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 [48] into Coq [18] types is formalized through an operator, written $\langle\cdot\rangle$, that maps all arrow types towards the type Func and maps all reference types towards the type Loc. A Caml value of type τ is thus represented as a Coq value of type $\langle\tau\rangle$. The definition of the operator $\langle\cdot\rangle$ is as follows.

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

On the one hand, ML typing is very useful as knowing the type of values allows to reflect Caml values directly into the corresponding Coq values. On the other hand, typing is restrictive: there are numerous programs that are correct but cannot be type-checked in ML. In particular, the ML type system does not accommodate programs involving null pointers and strong updates. Yet, although those features are difficult to handle in a type system without compromising type safety, their correctness can be justified through proofs of program correctness. So, I extend Caml with null pointers and strong udpates, and then use characteristic formulae to

justify that null pointers are never dereferenced and that reads in memory always yield a value of the expected type.

The correctness of this approach is however not entirely straightforward to justify. On the one hand, characteristic formulae are generated from typed programs. On the other hand, the introduction of null pointers and strong updates may jeopardize the validity of types. To justify the soundness of characteristic formulae, I introduce a new type system called weak-ML. This type system does not enjoy type soundness. However, it carries all the type information and invariants needed to generate characteristic formulae and to prove them sound.

In short, weak-ML corresponds to a relaxed version of ML that does not keep track of the type of pointers or functions, and that does not impose any constraint on the typing of dereferencing and applications. The translation from Caml types to Coq types is in fact conducted in two steps: a Caml type is first translated into a weak-ML type, and this weak-ML type it then translated into a Coq type.

Functions To specify the behavior of functions, I rely on a predicate called AppReturns. The proposition "$\mathsf{AppReturns}\, f\, v\, H\, Q$" 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 AppReturns to f and v.

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

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 type of AppReturns is as follows.

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

For example, the function incr is specified as shown below, using a pre-condition of the form $(l \hookrightarrow n)$ and a post-condition describing a heap of the form $(l \hookrightarrow n+1)$. Note: the abstraction "$\lambda\_.$" is used to discard the unit value returned by the function incr.

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

To establish a property about the behavior of an application, one needs to exploit an instance of AppReturns. Such instances can be derived from the characteristic formula associated with the definition of the function involved. If a function f is defined as the abstraction "$\lambda x.\, t$", then, given a particular argument x, one can derive an instance of "$\mathsf{AppReturns}\, f\, x\, H\, Q$" simply by proving that the body t admits the pre-condition H and the post-condition Q for that particular argument x. The

characteristic formula of a function definition ($\mathsf{let\,rec}\, f = \lambda x.\, t\, \mathsf{in}\, t'$) is defined as follows.

$$\begin{array}{l}[\![\mathsf{let\,rec}\, f = \lambda x.\, t\, \mathsf{in}\, t']\!] \quad \equiv \\ \quad \lambda H Q.\, \forall f.\, (\forall x\, H'\, Q'.\, [\![t]\!]\, H'\, Q' \Rightarrow \mathsf{AppReturns}\, f\, x\, H'\, Q') \Rightarrow [\![t']\!]\, H\, Q\end{array}$$

Observe that the formula does not involve a specific treatment of recursivity. Indeed, to prove that a recursive function satisfies a given specification, it suffices to conduct a proof by induction that the function indeed satisfies that specification. The induction may be conducted on a measure or on a well-founded relation, using the induction facility from the theorem prover. So, there is no need to strengthen the characteristic formula in any way for supporting recursive functions. A similar observation was also made by Honda *et al* [40].

1.3 Implementation

I have implemented a tool, called CFML, short for "Characteristic Formulae for ML", that targets the verification of Caml programs [48] using the Coq proof assistant [18]. This tool comprises two parts. The CFML generator is a program that translates Caml source code into a set of Coq definitions and axioms. The CFML library is a Coq library that contains notation, lemmas and tactics for manipulating characteristic formulae. In this section, I describe precisely the fragment of the OCaml language supported by CFML, as well as the exact logic in which the reasoning on characteristic formulae is taking place. I also give an overview of the architecture of CFML, and comment on its trusted computing base.

Source language I have focused on a subset of the OCaml programming language, which is a sequential, call-by-value, high-level programming language. The current implementation of CFML supports the core λ-calculus, including higher-order functions, recursion, mutual recursion and polymorphic recursion. It supports tuples, data constructors, pattern matching, reference cells, records and arrays. I provide an additional Caml library that adds support for null pointers and strong updates.

For the sake of simplicity, I model machine integers as mathematical integers, thus making the assumption that no arithmetic overflow ever occurs. Moreover, I do not include floating-point numbers, whose formalization is associated with numerous technical difficulties. Also, I assume the source language to be deterministic. This determinacy requirement might in fact be relaxed, but the formal developments were slightly simpler to conduct under this assumption.

Lazy expressions are supported under the condition that the code would terminate without any lazy annotation. Although this restriction does not enable reasoning on infinite data structures, it covers some uses of laziness, such as computation scheduling, which is exploited in particular in Okasaki's data structures [69]. In fact, CFML simply ignores any annotation relative to laziness. Indeed, if a program

satisfies its specification when evaluated without any lazy annotation, then it also satisfies its specification when evaluated with lazy annotations. (The reciprocal is not true.)

Moreover, CFML includes an experimental support for Caml modules and functors, which are reflected as Coq modules and functors. This development is still considered experimental because of several limitations of the current module system of Coq.[1] Yet, the support for modules has shown useful in the verification of data structures implemented using Caml functors.

Thereafter, I refer to the subset of the OCaml language supported by CFML as "Caml".

Target logic When verifying programs through characteristic formulae, both the specification process and the verification process take place in the logic. The logic targeted by CFML is the logic of Coq. More precisely, I work in the Calculus of Inductive Constructions, strengthened with the following standard extensions: functional extensionality, propositional extensionality, classical logic, and indefinite description (also called Hilbert's epsilon operator). Note that the proof irrelevance property is derivable in this setting.

Those axioms are all compatible with the standard boolean model of type theory. They are not included by default in Coq but they are integrated in other higher-order logic proof assistants such as Isabelle/HOL and HOL4. Remark: although the CFML library is implemented in the Coq proof assistant, the developments it contains could presumably be reproduced in another general-purpose proof assistant based on higher-order logic.

The CFML generator The CFML generator starts by parsing a Caml source file. This source code need not be annotated with any specification nor invariant. The tool then normalizes the source code in order to assign names to intermediate expressions, that is, to make sequencing explicit. It then type-checks the code and generates a Coq file. This Coq file contains a set of declarations reflecting every top-level declaration from the source code.

For example, consider a top-level function definition "$\mathsf{let}\, f\, x = t$". The CFML generator produces an axiom named f, of type Func, as well as an axiom named f_cf, which allows deriving instances of the predicate AppReturns for the function f.

$$\begin{array}{lcl} \mathsf{Axiom\ f} & : & \mathsf{Func}. \\ \mathsf{Axiom\ f\_cf} & : & \forall x\, H\, Q.\ [\![t]\!]\, H\, Q \Rightarrow \mathsf{AppReturns}\, \mathsf{f}\, x\, H\, Q. \end{array}$$

The specification and verification of the content of a file called demo.ml takes place in a proof script called demo_proof.v. The file demo_proof.v depends on the

[1] A major limitation of the Coq module system is that the positivity requirement for inductive definitions is based on a syntactic check that does not appropriately support abstract type constructors. In other words, Coq does not implement a feature equivalent to the variance constraints of Caml. Another inconvenience is that Coq module signatures cannot be described on-the-fly like in Caml; they have to be named explicitly.

generated file, which is called demo_ml.v. When the source file demo.ml is modified, the CFML generator needs to be executed again, producing an updated demo_ml.v file. The proof script from demo_proof.v, which records the arguments explaining why the previous version of demo.ml was correct, may need to be updated. To find out where modifications are required, it suffices to run the Coq proof assistant on the proof script until the first point where a proof breaks. One can then fix the proof script so as to reflect the changes made in the source code.

The CFML library The CFML library is made of three main parts. First, it includes a system of notation for pretty-printing characteristic formulae. Second, it includes a number of lemmas for reasoning on the specification of functions. Third, it includes the definition of high-level tactics that are used to efficiently manipulate characteristic formuale.

The CFML library is built upon three axioms. The first one asserts the existence of a type Func, which is used to represent functions. The second one asserts the existence of a predicate AppEval, a low-level predicate in terms of which the predicate AppReturns is defined. Intuitively, the proposition AppEval $f\,v\,h\,v'\,h'$ corresponds to the big-step reduction judgment for functions: it is equivalent to $(f\,v)_{/h} \Downarrow v'_{/h'}$. The third axiom asserts that AppEval is deterministic: in the judgment AppEval $f\,v\,h\,v'\,h'$, the last two arguments are uniquely determined by the first three. The library also includes axioms for describing the specification of the primitive functions for manipulating references. Through the proof of soundness, I establish that all those axioms can be given a sound interpretation.

Trusted computing base The framework that I have developed aims at proving programs correct. To trust that CFML actually establish the correctness of a program, one needs to trust that the characteristic formulae that I generate and that I take as axioms are accurate descriptions of the programs they correspond to, and that those axioms do not break the soundness of the target logic.

The trusted computing base of CFML also includes the implementation of the CFML generator, which I have programmed in OCaml, the implementation of the parser and of the type-checker of OCaml, and the implementation of the Coq proof assistant. To be exhaustive, I shall also mention that the correctness of the framework indirectly relies on the correctness of the complete OCaml compiler and on the correctness of the hardware.

The main priority of my thesis has been the development and implementation of a practical tool for program verification. For this reason, I have not yet invested time in trying to construct a *mechanized proof* of the soundness of characteristic formulae. In this dissertation, I only give a detailed *paper proof* justifying that all the axioms involved can be given a concrete interpretation in the logic. An interesting direction for future work consists in trying to replace the axioms with definitions and lemmas, which would be expressed in terms of a deep embedding of the source language.

1.4 Contribution

The thesis that I defend is summarized in the following statement.

> **Generating the characteristic formula of a program and exploiting that formula in an interactive proof assistant provides an effective approach to formally verifying that the program satisfies a given specification.**

A characteristic formula is a logical predicate that precisely describes the set of specifications admissible by a given program, without referring to the syntax of the programing language. I have not invented the general concept of characteristic formulae, however I have turned that concept into a practical program verification technique. More precisely, the main contributions of my thesis may be summarized as follows.

I show that characteristic formulae can be expressed in terms of a standard higher-order logic. Moreover, I show that characteristic formulae can be pretty-printed just like the source code they describe. Hence, compared with Honda et al's work, the characteristic formulae that I generate are easy to read and can be manipulated within an off-the-shelf theorem prover. I also explain how characteristic formulae can support local reasoning. More precisely, I rely on separation-logic style predicates for specifying memory states, and the characteristic formulae that I generate take into account potential applications of the frame rule.

I demonstrate the effectiveness of characteristic formulae through the implementation of a tool called CFML that accepts Caml programs and produces Coq definitions. I have used CFML to verify a collection of purely-functional data structures taken from Okasaki's book on purely-functional data structures [69]. Some advanced structures, such as bootstrapped queues, had never been formally verified previously.

I have investigated the treatment of higher-order imperative functions. In particular, I have verified the following functions: a higher-order iterator for lists, a function that manipulates a list of counters in which each counter is a function that carries its own private state, the generic combinator compose, and Reynolds' CPS-append function [77]. More recently, I have verified three imperative algorithms: an implementation of Dijkstra's shortest path algorithm (the version of the algorithm that uses a priority queue), an implementation of Tarjan's Union-Find data structure (specified with respect to a partial equivalence relations), and an implementation of sparse arrays (as described in the first task from the Vacid-0 challenge [78]). Those developments, which are not described in this manuscript, can be found online[2].

[2]Proof scripts: `http://arthur.chargueraud.org/research/2010/thesis/`

1.5 Research and publications

During my thesis, I have worked on three main projects. First, I have studied a type system based on linear capabilities for describing mutable state. This type system has been described in an ICFP'08 paper, and it is not included in my dissertation. Second, I have worked on a deep embedding of the pure fragment of Caml in Coq. This deep embedding is described in a research paper which has not been published. The work on the deep embedding led me to characteristic formulae for purely-functional Caml programs. This work has appeared as an ICFP'10 paper. Most of the content of that paper is included in my dissertation. I have recently extended characteristic formulae to imperative programs, reusing in particular ideas that had been developed for the type system based on capabilities. At the time of writing, the extension of characteristic formulae to the imperative setting has not yet been submitted as a conference paper.

Previous research papers:

- *Functional Translation of a Calculus of Capabilities,*
  Arthur Charguéraud and François Pottier,
  International Conference on Functional Programming (ICFP), 2008.
- *Verification of Functional Programs Through a Deep Embedding,*
  Arthur Charguéraud,
  Unpublished, 2009.
- *Program Verification Through Characteristic Formulae,*
  Arthur Charguéraud,
  International Conference on Functional Programming (ICFP), 2010.

1.6 Structure of the dissertation

The thesis is structured as follows. Examples of verification through characteristic formulae is described in Chapter 2 for pure programs, and in Chapter 3 for imperative programs. The construction of characteristic formulae for pure programs is developed in Chapter 4. This construction is then generalized to imperative programs in Chapter 5. The proof of soundness and completeness of characteristic formulae for pure programs is the matter of Chapter 6. Those proofs are then extended to an imperative setting in Chapter 7. Finally, related work is discussed in Chapter 8, and conclusions are given in Chapter 9.

Notice: several predicates are used with a different meaning in the context of purely-functional programs than in the context of imperative programs. For example, the big-step reduction judgment for pure programs takes the form $t \Downarrow v$, whereas the judgment for imperative programs takes the form $t_{/m} \Downarrow v'_{/m'}$. Similarly, reasoning on applications in a purely-functional setting involves the predicate

"AppReturns $f\ v\ P$", whereas in an imperative setting the same predicate takes the form "AppReturns $f\ v\ H\ Q$". This reuse of predicate names, which makes the presentation lighter and more uniform, should not lead to confusion since a given predicate is always used in a consistent manner in each chapter.

Chapter 2

Overview and specifications

The purpose of this overview is to give some intuition on how to construct a characteristic formula and how to exploit it. To that end, I consider a small but illustrative purely-functional function as running example. Another goal of the chapter is to explain the specification language upon which I rely. Specifications take the form of Coq lemmas, stated using special predicates as well as a layer of notation. The specification of modules and functors is illustrated through the presentation of a case study on purely-functional red-black trees.

2.1 Characteristic formulae for pure programs

In a purely-functional setting, the characteristic formulae $[\![t]\!]$ associated with a term t is such that, for any given post-condition P, the proposition "$[\![t]\!]\,P$" holds if and only if the term t terminates and returns a value satisfying the predicate P. Note that reasoning on the total correctness of pure programs can be conducted without pre-conditions. In terms of types, the characteristic formula associated with a term t of type τ applies to a post-condition P of type $\langle\tau\rangle \to \mathsf{Prop}$ and produces a proposition, so $[\![t]\!]$ admits the type $(\langle\tau\rangle \to \mathsf{Prop}) \to \mathsf{Prop}$. In terms of a denotational interpretation, $[\![t]\!]$ corresponds to the set of post-conditions that are valid for the term t. I next describe an example of a characteristic formula.

2.1.1 Example of a characteristic formula

Consider the following recursive function, which divides by two any non-negative even integer. For the interest of the example, the function diverges when called on a negative integer and crashes when called on an odd integer. Through this example, I present the construction of characteristic formulae and also illustrate the treatment

of partial functions, of recursion, and of ghost variables.

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

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

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

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

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

$$
\begin{array}{l}
\mathbf{Let}\ \mathsf{half} = \mathbf{fun}\ x \mapsto \\
\quad \mathbf{if}\ x = 0\ \mathbf{then\ return}\ 0 \\
\quad \mathbf{else\ if}\ x = 1\ \mathbf{then\ crash} \\
\quad \mathbf{else\ let}\ y = \mathbf{app}\ \mathsf{half}\,(x-2)\ \mathbf{in} \\
\qquad \mathbf{return}\ (y+1)
\end{array}
$$

At first sight, it might appear that the characteristic formula is merely a rephrasing of the source code in some other syntax. To some extent, this is true. A characteristic formula is a sound and complete description of the behavior of a program.

Thus, it carries no more and no less information than the source code of the program itself. However, characteristic formulae enable us to move away from program syntax and conduct program verification entirely at the logical level. Characteristic formulae thereby avoid all the technical difficulties associated with manipulation of program syntax and make it possible to work directly in terms of higher-order logic values and formulae.

2.1.2 Specification and verification

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

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

Observe a function f is described in Coq at the type Func, regardless of Caml type of the function f. Having Func as a constant type and not a parametric type allows for a simple treatment of polymorphic functions and of functions that admit a recursive type.

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

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

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

The next step consists in proving that the function half satisfies its specification. This is done by exploiting its characteristic formula. I first give the mathematical presentation of the proof and then show the corresponding Coq proof script. The specification is proved by induction on x. Let x and n be such that $n \geq 0$ and $x = 2 * n$. We apply the characteristic formula to prove "$\mathsf{AppReturns}\, \mathsf{half}\, x\, (= n)$". If x is equal to 0, we conclude by showing that n is equal to 0. If x is equal to 1, we show that $x = 2 * n$ is absurd. Otherwise, $x \geq 2$. We instantiate P' as "$= n - 1$",

and prove "$\mathsf{AppReturns}\,\mathsf{half}\,(x-2)\,P'$" using the induction hypothesis. Finally, we show that, for any y such that $y = n - 1$, the proposition $y + 1 = n$ holds. This completes the proof. Note that, through this proof by induction, we have proved that the function half terminates when it is applied to a non-negative even integer.

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

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

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

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

```
x : int
IH : forall x', 0 <= x' -> x' < x ->
     forall n, n >= 0 -> x' = 2 * n ->
     AppReturns half x' (= n)
n : int
Pos : n >= 0
```

```
Eq : x = 2 * n
C1 : x <> 0
C2 : x <> 1
------------------------------------------------
(Let y := App half (x-2) in Return (1+y)) (= n)
```

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

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

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

Automation, denoted by the tilde symbol, is able to handle all the subgoals produced by `xgo`.

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

To summarize, my approach allows for very concise proof scripts whenever verifying simple pieces of code, thanks to the automated processing done by `xgo` and

[1] The name `y` is a bound name in the characteristic formula. Since Coq tactics are not allowed to depend on bound names, the tactic `xgo` actually takes as argument a constant called `'y`. The constant `'y`, defined by the CFML generator, serves as an identifier used to tag the characteristic formula associated with the subterm "$\mathsf{let}\, y = \mathsf{half}\,(x-2)\,\mathsf{in}\, y+1$". This tagged formula is displayed in Coq as follows: `Let 'y y := App half (x-2) in Return (1+y)`. If the tactic language did support ways of referring to bound variables, then the work-around described in this footnote would not be needed.

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

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

Figure 2.1: Module signatures for finite sets and ordered types

to the good amount of automation available through the proof search mechanism and the decision procedures that can be called from Coq. At the same time, when verifying more complex code, my approach offers a very fine-grained control on the structure of the proofs and it greatly benefits from the integration in a proof assistant for proving nontrivial facts interactively.

2.2 Formalizing purely functional data structures

Chris Okasaki's book *Purely Functional Data Structures* [69] contains a collection of efficient data structures, with concise implementation and nontrivial invariants. Its code appeared as a excellent benchmark for testing the usability of characteristic formulae for verifying pure programs. I have verified more than half of the contents of the book. Here, I focus on describing the formalization of red-black trees and give statistics on the other formalizations completed.

Red-black trees are binary search trees where each node is tagged with a color, either red or black. Those tags are used to maintain balance in the tree, ensuring a logarithmic asymptotic complexity. Okasaki's implementation appears in Figure 2.2. It consists of a functor that, given an ordered type, builds a module matching the signature of finite sets. The Caml signatures involved appear in Figure 2.1.

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

2.2.1 Specification of the signature

In order to specify functions manipulating red-black trees, I introduce a representation predicate called rep. Intuitively, every data structure admits a mathematical model. For example, the model of a red-black tree is a set of values. Similarly, the model of a priority queue is a multiset, and the model of a queue is a sequence (a list). Sometimes, the mathematical model is simply the value itself. For instance, the model of an integer or of a value of type color is just the value itself.

```
module RedBlackSet (Elem : Ordered) : Fset = struct

   type elem = Elem.t
   type color = Red | Black
   type fset = Empty | Node of color * fset * elem * fset

   let empty = Empty

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

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

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

end
```

Figure 2.2: Okasaki's implementation of Red-Black sets

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

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

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

Remark: although representation predicates have appeared previously in the context of interactive program verification (e.g. [63, 27, 56]), my work seems to be the first to use them in a systematic manner through a typeclass definition.

Figure 2.3 contains the specification for an abstract finite set module named F. Elements of the sets, of type elem, are expected to be modelled by some type T and to be related to their models by an instance of type "Rep elem T". Moreover, the values implementing finite sets, of type fset, should be related to their model, of type set T, through an instance of type "Rep fset (set T)". The module signature then contains the specification of the values from the finite set module F. The first one asserts that the value empty should be a representation for the empty set. The specifications for insert and member rely on a special notation, explained next.

So far, I have relied on the predicate AppReturns to specify functions. Even though AppReturns works well for functions of one argument, it becomes impractical for curried functions of higher arity, in particular because one needs to specify the behavior of partial applications. So, I introduce the Spec notation, explaining its meaning informally and postponing its formal definition to §4.2.3. With the Spec notation, the specification of insert, shown below, reads like a prototype: insert takes two arguments, x of type elem and e of type fset. Then, for any model X of x and for any set E that models e, the function returns a finite set e' which admits a model E' equal to {X} ∪ E. Below, \{X} is a Coq notation for a singleton set and \u denotes the set union operator.

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

```
      rep e' E' /\ E' = \{X} \u E).
```

The conclusion, which takes the form "`R (fun e' => H)`", can be read as "the application of `insert` to the arguments `x` and `e` returns a value `e'` satisfying `H`". Here `R` is bound in the notation "`|R»`". The notation will be explained later on (§4.2.2).

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

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

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

The specification for the function insert given in Figure 2.3 makes two further simplifications. First, it relies on the notation RepTotal, which avoids the introduction of a name R when it is immediately applied. Second, it makes uses of a partial application of equality, of the form "$= \{\mathsf{X}\} \cup \mathsf{E}$", for the sake of conciseness. Overall, the interest of introducing several layers of notation is that the final specifications from Figure 2.3 are about the most concise formal specifications one could hope for.

Let me briefly describe the remaining specifications. The function member takes as argument a value x and a finite set e, and returns a boolean which is true if and only if the model X of x belongs to the model E of e. Figure 2.4 contains the specification of an abstract ordered type module named O. Elements of the ordered type t should be modelled by a type T. Values of type T should be ordered by a total order relation. The order relation and the proof that it is total are described through instances of the typeclasses Le and Le_total_order, respectively. An instance of the strict-order relation (LibOrder.lt) is automatically derived through the typeclass mechanism. This relation is used to specify the boolean comparison function lt, defined in the module O.

2.2.2 Verification of the implementation

It remains to specify and verify the implementation of red-black trees. Consider a module O that describes an ordered type. Assume the module O has been verified through a Coq module named OS of signature OrderedSigSpec. Our goal is then to prove correct the module obtained by applying the functor RedBlackSet to the module O, through the construction of a Coq module of signature FsetSigSpec. Thus, the verification of the Caml functor RedBlackSet is carried through the implementation of a Coq functor named RedBlackSetSpec, which depends both on the module O and on its specification OS. For the Coq experts, I show the first few lines of this Coq functor are shown below

```
Module Type FsetSigSpec.

   Declare Module F : MLFset. Import F.

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

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

End FsetSigSpec.
```

Figure 2.3: Specification of finite sets

```
Module Type OrderedSigSpec.

   Declare Module O : MLOrdered. Import O.

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

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

End OrderedSigSpec.
```

Figure 2.4: Specification of ordered types

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

The next step in the construction of this functor is the definition of an instance of the representation predicate for red-black trees. To start with, assume that our goal is simply to specify a binary search tree (not necessarily balanced). The rep predicate would be defined in terms of an inductive invariant called inv, as shown below. First, inv relates the empty tree to the empty set. Second, inv relates a node with root y and subtrees a and b to the set $\{Y\} \cup A \cup B$, where the uppercase variables are the models associated with their lowercase counterpart. Moreover, we need to ensure that all the elements of the left subtree A are smaller than the root Y, and that, symmetrically, elements from B are greater than Y. Those invariants are stated with help of the predicate foreach. The proposition "foreach P E" asserts that all the elements in the set E satisfy the predicate P.

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

A red-black tree is a binary search tree satisfying three additional invariants. First, every path from the root to a leaf should contain the same number of black nodes. Second, no red node can have red child. Third, the root of the entire tree must be black. In order to capture the first invariant, I extend the predicate inv so that it depends on a natural number n representing the number of black nodes to be found in every path. For an empty tree, this number is zero. For a nonempty tree, this number is equal to the number m of black nodes that can be found in every path of each of the two subtrees, augmented by one if the node is black. The second invariant, asserting that a red node must have black children, can be enforced simply by testing colors. The rep predicate then relates a red-black tree e with a set E if there exists a value n such that "inv n e E" holds and such that the root of e is black (the third invariant). The resulting definition of inv appears in Figure 2.5.

In practice, I further extend the invariant with an extra boolean (this extended definition does not appear here; it can be found in the Coq development). When the boolean is true, the definition of inv is unchanged. However, when the boolean is false, then the second invariant, which asserts that a red node cannot have a red children, might be broken at the root of the tree. This relaxed version of the invariant is useful to specify the behavior of the auxiliary function balance. Indeed, this function takes as input a color, an item and two subtrees, and one of those two subtrees might have its root incorrectly colored.

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

Global Instance fset_rep : Rep fset (set T).
Proof. apply (Build_Rep
  (fun e E => exists n, inv n e E /\ root_color e = Black)).
  ... (* the proof for the field rep_unique is not shown *)
Defined.
```

Figure 2.5: Representation predicate for red-black trees

Figure 2.6 shows the lemma corresponding to the verification of insert. Observe that the local recursive function ins is specified in the script. It is then verified with the help of the tactic xgo.

2.2.3 Statistics on the formalizations

I have specified and verified various implementations of queues, double-ended queues, priority queues (heaps), sets, as well as sortable lists, catenable lists and random-access lists. Caml implementations are directly adapted from Okaski's SML code [69]. All code and proofs can can be found online.[2] Figure 2.7 contains statistics on the number of non-empty lines in Caml source code and in Coq scripts. The programs considered are generally short, but note that Caml is a concise language and that Okasaki's code is particularly minimalist. Details are given about Coq scripts.

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

I evaluate the relative cost of a formal verification by comparing the number

[2] http://arthur.chargueraud.org/research/2010/cfml/

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

Figure 2.6: Invariant and model of red-black trees

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

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

Figure 2.7: Non-empty lines of source code and proof scripts

Chapter 3

Verification of imperative programs

This chapter contains an overview of the treatment of imperative programs. After introducing some notation for specifying heaps, I present the specification of the primitive functions for manipulating references, and I explain the treatment of for-loops and while-loops, using loop invariants. I then introduce representation predicates for mutable data structures, focusing on mutable lists to illustrate the definition and manipulation of such predicates. I study the example of a function that computes the length of a mutable list and explain why reasoning on loops using invariants does not take advantage of local reasoning. I then present a different treatment of loops. Finally, I explain the treatment of higher-order functions and that of functions with local state.

3.1 Examples of imperative functions

3.1.1 Notation for heap predicates

I start by describing informally the combinators for constructing heap predicates, which have type Hprop. Formal definitions are postponed to §5.2.1. The empty heap is written [] and the spatial conjunction of two heaps is written $H_1 * H_2$. The predicate "$l \hookrightarrow_{\mathcal{T}} v$" describes a reference cell l whose contents is described in Coq as the value v of type $\mathcal{T}$. Since the type $\mathcal{T}$ can be deduced from the value v, I often omit it and write "$l \hookrightarrow v$". A more general predicate for describing mutable data structures, written "$l \rightsquigarrow T\,v$", is explained further on. I lift propositions and existentials into heap predicates as follows. The predicate $[P]$ holds of the empty heap when the proposition P is true. The predicate $\exists x.\, H$ holds of a heap h if there exists a value x such that the predicate H holds of h (note that x is bound in H). The corresponding ASCII notations, which are used in Coq statements, appear below.

Heap predicate	Latex notation	ASCII notation
Empty heap	$[]$	`[]`
Separating conjunction	$H_1 * H_2$	`H1 \* H2`
Mutable data structure	$l \rightsquigarrow T\, v$	`l ~> T v`
Reference cell	$l \hookrightarrow v$	`l ~~> v`
Lifted proposition	$[P]$	`[P]`
Lifted existential	$\exists x.\, H$	`Hexists x, H`

The post-condition for a term t describes both the output value and the output state returned by the evaluation of t. If t admits the type τ, then the post-condition for its evaluation is a predicate of type $\langle\tau\rangle \to \mathsf{Hprop}$. So, a post-condition generally takes the form $\lambda x.\, H$. In the particular case where the return value x is of type unit, the name x is irrelevant, and I write $\#\, H$ as a shorthand for $\lambda\_ : \mathsf{unit}.\, H$. In the particular case where the evaluation of a term returns exactly a value v in the empty heap, the post-condition takes the form $\lambda x.\, [x = v]$. To describe such a post-condition, I use the shorthand $\backslash{=}\, v$. Finally, it is useful to extend a post-condition with a piece of heap. The predicate $Q \star H$ is a post-condition for a term that returns a value x and a heap h such that the heap h satisfies the predicate $(Q\, x) * H$. The corresponding ASCII notations are given in the next table.

Post-condition predicate	Latex notation	ASCII notation
General post-condition	$\lambda x.\, H$	`fun x => H`
Unit return value	$\#\, H$	`# H`
Exact return value	$\backslash{=}\, v$	`\= v`
Separating conjunction	$Q \star H$	`Q \*+ H`

The symbol ($\rhd$) denotes entailment between heap predicates, so the proposition $H_1 \rhd H_2$ asserts that any heap satisfying H_1 also satisfies H_2. Similarly, $Q_1 \blacktriangleright Q_2$ asserts that a post-condition Q_1 entails a post-condition Q_2, in the sense that for any value x and any heap h, the proposition $Q_1\, x\, h$ implies $Q_2\, x\, h$.

Entailment relation	Latex notation	ASCII notation
Between heap predicates	$H_1 \rhd H_2$	`H1 ==> H2`
Between post-conditions	$Q_1 \blacktriangleright Q_2$	`Q1 ===> Q2`

3.1.2 Specification of references

In this section, I present the specification of functions manipulating references. I start with the function incr, which increments the contents of a reference on an integer. Recall the specification of the function incr from the introduction. It asserts that the proposition "$\mathsf{AppReturns}\, \mathsf{incr}\, r\, (r \hookrightarrow n)\, (\#\, r \hookrightarrow n{+}1)$" holds for any location r and any integer n. The specification can be presented using the notation Spec, which is like that introduced in the previous chapter except that the variable R bound by the Spec notation now stands for a predicate that takes as argument both a pre-condition and a post-condition.

```
Lemma incr_spec :
  Spec incr r |R>>  forall n,  R (r ~~> n) (# r ~~> n+1)
```

Observe how the variable R, which describes the behaviour of the application of incr to the argument r, plays the same role as the predicate "AppReturns incr r".

The primitive function ref takes as argument a value v and allocates a reference with contents v. Its pre-condition is the empty heap, and its post-condition asserts that the application of ref to v produces a location l in a heap of the form $l \hookrightarrow v$. The specification is polymorphic in the type A of the value v. Remark: since ref is a built-in function, its specification is not proved as a lemma but is taken as an axiom.

```
Axiom ref_spec : forall A,
  Spec ref (v:A) |R>>  R [] (fun l => l ~~> v).
```

The function get applies to a location l and to a heap of the form $l \hookrightarrow v$, and returns exactly the value v, without changing the heap. Hence its specification shown next.

```
Spec get (l:Loc) |R>> forall (v:A),
  R (l ~~> v) (\=v \*+ l ~~> v).
```

The function set applies to a location l and to a value v. The heap must be of the form $l \hookrightarrow v'$ for some value v', indicating that the location l is already allocated. The application of set produces the unit value and a heap of the form $l \hookrightarrow v$.

```
Spec set (l:Loc) (v:A) |R>> forall (v':A),
  R (l ~~> v') (# l ~~> v).
```

One can prove the specification of incr with respect to the specifications of get and set. The verification is conducted on the administrative normal form of the definition of the function incr, where the side-effect associated with reading the contents of the reference is separated from the action of updating the contents of the reference. The normal form of incr, shown next, is automatically generated by CFML.

```
let incr r = (let x = get r in set r (x + 1))
```

The proof of the specification of incr is quite short: "`xgo.xsimpl~.`". The tactic `xgo` follows the structure of the code and exploits the specifications of the functions for reading and updating references. The tactic `xsimpl~` is used to discharge the proof obligation produced, which simply consists in checking that, under the assumption $x = n$, the heap predicate $r \hookrightarrow x + 1$ is equivalent to the heap predicate $r \hookrightarrow n + 1$.

3.1.3 Reasoning about for-loops

To illustrate the treatment of for-loops, I consider a function that takes as argument a non-negative integer k and a reference r, and then calls k times the function incr on the reference r.

```
let incr_for k r = (for i = 1 to k do (incr r) done)
```

The specification of this function asserts that if n is the initial contents of the reference r, then $n + k$ is its final content.

```
Spec incr_for k r |R>> forall n, k >= 0 ->
  R (r ~~> n) (# r ~~> n + k).
```

The proof involves providing a loop invariant for reasoning on the loop. The tactic `xfor_inv` expects a predicate I, of type $\mathsf{Int} \to \mathsf{Hprop}$, describing the state of the heap in terms of the loop counter. In the example, the appropriate loop invariant, called I, is defined as $\lambda i.(r \hookrightarrow n + i - 1)$. One may check that an execution of the loop body, which increments r, turns a heap satisfying $I\,i$ into a heap satisfying $I\,(i+1)$. One may also check that, when i is equal to 1, the heap predicate $I\,1$ is equivalent to the initial heap $(r \hookrightarrow n)$. Moreover, when i is equal to $k+1$, the heap predicate $I\,(k+1)$ is equivalent to the final heap $(r \hookrightarrow n + k)$. Note that the post-condition of the loop is described by $I\,(k+1)$ and not by $I\,k$, because $I\,k$ describes the heap *before* the final iteration of the loop, whereas $I\,(k+1)$ describes the heap *after* that final iteration.

The characteristic formula of a for-loop of the form "$\mathsf{for}\,i = a\,\mathsf{to}\,b\,\mathsf{do}\,t$" is shown next. The invariant I is quantified existentially at head of the formula. A conjunction of three propositions follows. The first one asserts that the initial heap H entails the application of the invariant to the initial value a of the loop counter. The second one asserts that the execution of the body t of the loop, starting from a heap satisfying the invariant $I\,i$ for a valid value of the loop counter i, terminates and produces a heap that satisfies the invariant $I\,(i+1)$. The third and last proposition checks that the predicate $I\,(b+1)$ entails the expected final heap description $Q\,tt$ (where tt denotes the unit value). This third proposition is in fact generalized so as to correctly handle the case where a is greater than b, in which case the loop body is not executed at all. The heap predicate $I\,(\mathsf{max}\,a\,(b+1))$ concisely takes into account both the case $a \leq b$ and the case $a > b$.

$$[\![\mathsf{for}\,i = a\,\mathsf{to}\,b\,\mathsf{do}\,t]\!] \;\equiv\; \mathsf{frame}\,(\lambda HQ.\;\exists I.\;\begin{cases} H \rhd I\,a \\ \forall i \in [a,b].\;[\![t]\!]\,(I\,i)\,(\#\,I\,(i+1)) \\ I\,(\mathsf{max}\,a\,(b+1)) \rhd Q\,tt \end{cases})$$

Remark: the function that increments k times a reference r may in fact be assigned a more general specification that also covers the case where k is a negative value. This specification, shown next, involves a post-condition asserting that the contents of r is equal to n plus the maximum of k and 0.

```
Spec incr_for k r |R>> forall n,
  R (r ~~> n) (# r ~~> n + max k 0).
```

3.1.4 Reasoning about while-loops

I now adapt the previous example to a while-loop. The function shown below takes as argument two references, called s and r, and executes a while-loop. As long as

the contents of s is positive, the contents of r is incremented and the contents of s is decremented.

```
let incr_while s r = (while (!s > 0) do (incr r; decr s) done)
```

If k denotes the initial contents of s (assume k to be non-negative for the sake of simplicity) and if n denotes the initial contents of s, then the final contents of s is equal to zero and the final contents of r is equal to $n + k$. The corresponding specification appears next.

```
Spec incr_while s r |R>> forall k n, k >= 0 ->
   R (s ~~> k \* r ~~> n) (# s ~~> 0 \* r ~~> n+k).
```

The function is proved correct using a loop-invariant. Contrary to for-loops, there is no loop counter to serve as index for the invariant. So, I artificially introduce some indices. Those indices are used to describe how the heap evolves through the execution of the loop, and to argue for termination, using a well-founded relation over indices.

In the example, we may choose to index the invariant with values of types Int. The invariant I is then defined as "$\lambda i.\,(s \hookrightarrow i) * (r \hookrightarrow n + k - i) * [i \geq 0]$". The execution of the loop increments r and decrements s, so it turns a heap satisfying $I\,i$ into a heap satisfying $I\,(i-1)$. Observe that the initial state of the loop is described by $I\,k$ and that the final state of the loop is described by $I\,0$. Since the value of the index i decreases from k to 0 during the execution of the loop, the absolute value of i is a measure that justifies the termination of the loop.

The characteristic formula for a loop "$\mathsf{while}\,t_1\,\mathsf{do}\,t_2$", in its version based on a loop invariant, appears below. This formula quantifies existentially over a type A used to represent indices, over an invariant I of type $A \to \mathsf{Hprop}$ describing heaps before evaluating the loop condition, over an invariant J of type $A \to \mathsf{bool} \to \mathsf{Hprop}$ describing the post-condition of the loop condition (J is needed because the evaluation of t_1 may modify the store), and over a binary relation ($\prec$) of type $A \to A \to \mathsf{Prop}$.

A conjunction of five propositions follows. This first one asserts that ($\prec$), the relation used to argue for termination, is well-founded. The second one requires the existence of a index X_0 such that the initial heap satisfies the heap predicate $I\,X_0$. The third proposition states that, for any index X, the loop condition t_1 should be executable in a heap of the form $I\,X$ and produce a heap of the form $J\,X$. The two remaining proposition correspond to the cases where the loop condition returns true or false, respectively. When the loop condition returns true, the loop body t_2 is executed in a heap satisfying $J\,X\,\mathsf{true}$. The execution of that body should produce a heap satisfying the invariant $I\,Y$ for some index Y, which is existentially-quantified in the formula. To ensure termination, Y has to be smaller than X, written $Y \prec X$. When the loop condition returns false, the loop terminates in a heap satisfying

$J\,X\,\mathsf{false}$. This heap description should entail the expected heap description $Q\,tt$.

$$[\![\mathsf{while}\,t_1\,\mathsf{do}\,t_2]\!] \equiv \mathsf{frame}\,(\lambda H Q.\ \exists A.\ \exists I.\ \exists J.\ \exists(\prec).\ \begin{cases} \mathsf{well\text{-}founded}(\prec) \\ \exists X_0.\ H \rhd I\,X_0 \\ \forall X.\ [\![t_1]\!]\,(I\,X)\,(J\,X) \\ \forall X.\ [\![t_2]\!]\,(J\,X\,\mathsf{true})\,(\#\,\exists Y.\ (I\,Y) * [Y \prec X]) \\ \forall X.\ J\,X\,\mathsf{false} \rhd Q\,tt \end{cases})$$

3.2 Mutable data structures

In this section, I introduce a family of predicates, written $x \rightsquigarrow G\,X$, for describing "linear" mutable data structures, such as mutable lists and mutable trees. More generally, a linear data structure is a structure that does not involve sharing of sub-structures. Those predicates are also used to describe purely-functional data structures that may contain mutable elements. The treatment of non-linear data structure, where pointer aliasing is involved, is discussed as well.

3.2.1 Recursive ownership

In the heap predicate $x \rightsquigarrow G\,X$, the value x denotes the entry point of a mutable data structure, for example the location at the head of a mutable list. The value X describes the mathematical model of that data structure, for example X might be a Coq list. The predicate G captures the relation between the entry point x of the structure, the mathematical model X of the structure, and the piece of heap containing the representation of that structure. The predicate G is called a representation predicate.

The predicate $x \rightsquigarrow G\,X$ is defined as $G\,X\,x$. (More generally, $x \rightsquigarrow P$ is defined as $P\,x$.) So, if x has type a and X has type A, then G has type $A \to a \to \mathsf{Hprop}$. The tagging of the application of $G\,X$ to x with an arrow symbol serves two purposes: improving the readability of specifications, and simplifying the implementation of the Coq tactics from the CFML library.

For example, the representation predicate Mlist is used to describe mutable lists. The heap predicate $l \rightsquigarrow \mathsf{Mlist}\,G\,L$ describes a heap that contain the representation of a mutable list starting at location l and whose mathematical model is the Coq list L. The representation predicate G here describes the relation between the items found in the mutable list and their mathematical model. So, when G has type $a \to A \to \mathsf{Hprop}$, the representation predicate $\mathsf{Mlist}\,G$ has type $\mathsf{Loc} \to \mathsf{List}\,A \to \mathsf{Hprop}$.

The representation predicate for pure values (e.g., values of type Int) simply asserts that the mathematical model of a base value is the value itself. This representation predicate is called Id, and its formal definition is shown next.

```
Definition Id (A:Type) (X:A) (x:A) : Hprop :=  [x = X].
```

Thereafter, I write Id_A to denote the type application of Id to the type A, and simply write Id when the type A can be deduced from the context. Note that, for any type A and any value x of type A, the predicate $x \rightsquigarrow \mathsf{Id}_A\, x$ holds of the empty heap.

Combining the predicate Mlist and Id, we can build for instance a heap predicate $l \rightsquigarrow \mathsf{Mlist}\, \mathsf{Id}_{\mathsf{Int}}\, L$, which describes a mutable list of integers. Here, the location l has type Loc and L has type $\mathsf{List}\, \mathsf{Int}$. More interestingly, we can go one step further and construct a heap predicate describing a mutable list that contains mutable lists of integers as elements, through a heap predicate of the form $l \rightsquigarrow \mathsf{Mlist}\, (\mathsf{Mlist}\, \mathsf{Id}_{\mathsf{Int}})\, L$. The mathematical model L here is a list of lists, of type $\mathsf{List}\, (\mathsf{List}\, \mathsf{Int})$. Intuitively, the representation predicate $\mathsf{Mlist}\, (\mathsf{Mlist}\, \mathsf{Id}_{\mathsf{Int}})$ describes the ownership of a mutable list and of all the mutable lists that it contains. There are, however, some situations where we only want to describe the ownership of the main mutable list. In this case, we view the mutable list as a list of locations, through a predicate of the form $l \rightsquigarrow \mathsf{Mlist}\, \mathsf{Id}_{\mathsf{Loc}}\, L$. The list L now has type $\mathsf{List}\, \mathsf{Loc}$, and the locations contained in L are entry points for other mutable lists, which can be described using other heap predicates.

To summarize, representation predicates for mutable data structures not only relate pieces of data laid out in memory with their mathematical model, they also indicate the extent of ownership. On the one hand, the application of a representation predicate to another nontrivial representation predicate describes recursive ownership of data structures. On the other hand, the representation predicate $\mathsf{Id}_{\mathsf{Loc}}$ "cuts" recursive ownership. Although representation predicates are commonplace in program verification, the definition of parametric representation predicates, where a representation predicate for a container takes as argument the representation predicate for the elements, does not appear to have been exploited in a systematic manner in other program verification tools. Remark: the polymorphic representation predicates that I use here are the heap-predicate counterpart of the capabilities involved in the type system that I have developed during the first year of my PhD, together with François Pottier [16].

3.2.2 Representation predicate for references

In this section, I explain how to define a representation predicate for references. I also describe the focus and unfocus operations, which allow rearranging one's view on memory.

So far, a heap containing a reference has been specified using the predicate $r \hookrightarrow x$, which asserts that the location r points towards the value x. To concisely describe the situation where x corresponds to the entry point of a mutable data structure, I introduce a representation predicate for references, called Ref. The heap predicate $r \rightsquigarrow \mathsf{Ref}\, G\, X$ describes a heap made of a memory cell at address r that points to some value x on the one hand, and of a heap described by the predicate $x \rightsquigarrow G\, X$ on the other hand.

```
Definition Ref (a:Type) (A:Type) (G:A->a->Hprop) (X:A) (r:Loc) :=
   Hexists x:a, (r ~~> x) \* (x ~> G X).
```

For example, a reference r whose contents is an integer n is described by the heap predicate $r \rightsquigarrow \mathsf{Ref}\,\mathsf{Id}_{\mathsf{Int}}\,n$, which is logically equivalent to $r \hookrightarrow_{\mathsf{Int}} n$. A more interesting example is that of a reference r whose content is the location of another reference, call it s, whose contents is the integer n. A direct description involves a conjunction of two heap predicates: $(r \rightsquigarrow \mathsf{Ref}\,\mathsf{Id}_{\mathsf{Loc}}\,s) * (s \rightsquigarrow \mathsf{Ref}\,\mathsf{Id}_{\mathsf{Int}}\,n)$. We may also describe the same heap as: $r \rightsquigarrow \mathsf{Ref}\,(\mathsf{Ref}\,\mathsf{Id}_{\mathsf{Int}})\,n$. This second predicate is more concise than the first one, however it does not expose the location s of the inner reference.

The heap predicate "$\exists s.\,(r \rightsquigarrow \mathsf{Ref}\,\mathsf{Id}_{\mathsf{Loc}}\,s) * (s \rightsquigarrow \mathsf{Ref}\,\mathsf{Id}_{\mathsf{Int}}\,n)$" and the heap predicate "$\mathsf{Ref}\,(\mathsf{Ref}\,\mathsf{Id}_{\mathsf{Int}})$" are equivalent in the sense that one can convert from one to the other by unfolding the definition of the representation predicate Ref. The action of unfolding the definition of Ref is called "focus", following a terminology introduced by Fähndrich and DeLine [23] and reused in [16] for a closely-related operation. The statement of the focus lemma for references is an implication between heap predicates. It takes the ownership of a reference r and of its contents represented by a model X with respect to some representation predicate G, and turns it into the conjunction of the ownership of a reference r that contains a base value x on the one hand, and of the ownership of a data structure whose entry point is x and whose model with respect to G is the value X on the other hand.

```
Lemma focus_ref : forall a A (G:A->a->Hprop) (r:Loc) (X:A),
  (r ~> Ref G X) ==> Hexists x, (r ~~> x) \* (x ~> G X).
```

The reciprocal operation is called "unfocus". It consists in packing the ownership of a reference with the ownership of its contents back into a single heap predicate. The unfocus lemma could be formulated as the reciprocal of the focus lemma, yet in practice it is more convenient to extract the existential quantifiers as shown next.

```
Lemma unfocus_ref : forall a A G (r:Loc) (x:a) (X:A),
  (r ~~> x) \* (x ~> G V) ==> (r ~> Ref G X).
```

The lemmas `focus_ref` and `unfocus_ref` are involved in verification proof scripts for changing from one view on memory to another. In the current implementation, those lemmas need to be invoked explicitly by the user, yet it might be possible to design specialized decision procedures that are able to automatically exploit focus and unfocus lemmas.

3.2.3 Representation predicate for lists

In this section, I present the definition of the representation predicate Mlist for mutable lists, as well as the representation predicate Plist for purely-functional lists. I then describe the focus and unfocus lemmas associated with list representation predicates.

For the sake of type soundness, the Caml programming language does not feature null pointers. Nevertheless, a standard backdoor (called "Obj.magic") can be exploited to augment the language with null pointers. In Caml extended with null

pointers, one can implement C-style mutable lists. The head cell of a nonempty mutable list is represented by a reference on a pair of an item and of the tail of the list (a two-field record could also be used), and the empty list is represented by the null pointer. Hence the Caml type of mutable list shown next.

```
type 'a mlist = ('a * 'a mlist) ref
```

The Coq definition of the representation predicate Mlist appears next. It relates a pointer l with a Coq list L. The predicate Mlist is defined inductively following the structure of the list L, as done in earlier work on Separation Logic. When L is the empty list, the pointer l should be the null pointer, written `null`. When L has the form $X :: L'$, the pointer l should point to a reference cell that contains a pair (x, l'), such that X is the model of the data structure of entry point x and such that L' is the model of the list starting at pointer l'. Note that a heap predicate of the form $(l \hookrightarrow v)$ entails the fact that the location l is not null.

```
Fixpoint Mlist A a (G:A->a->Hprop) (L:list A) (l:Loc) : Hprop :=
  match L with
  | nil => [l = null]
  | X::L' => Hexists x l',
              (l ~~> (x,l')) \* (x ~> G X) \* (l' ~> Mlist G L')
  end.
```

One can devise a slightly more elegant definition of Mlist using the representation predicate Ref and the representation predicate Pair, which is explained next. The definition of Pair is such that a pair (X, Y) is the mathematical model of a pair (x, y) with respect to the representation predicate "Pair G_1 G_2" when X is the model of x with respect to G_1 and Y is the model of y with respect to G_2. The Coq definition of Pair appears next (for the sake of readability, I pretend that a Coq definition can bind pairs of names directly).

```
Definition Pair A B a b (G1:A->a->Hprop) (G2:B->b->hprop)
                        ((X,Y):A*B) ((x,y):a*b) : Hprop :=
  (x ~> G1 X) \* (y ~> G2 Y).
```

The definition of Mlist can now be improved as follows. The list $X :: L'$ is the model of the pointer l with respect to the representation predicate Mlist G if the pair (X, L') is the mathematical model of the pointer l with respect to the representation predicate Ref (Pair G (Mlist G)). So, the updated definition no longer needs to involve existential quantifiers. Existential quantifiers are only involved in the focus lemma presented further on.

```
Fixpoint Mlist A a (G:A->a->Hprop) (L:list A) (l:Loc) : Hprop :=
  match L with
  | nil => [l = null]
  | X::L' => l ~> Ref (Pair G (Mlist G)) (X,L')
  end.
```

I now define the representation predicate Plist for purely-functional lists. Contrary to Mlist, the predicate Plist relates a Coq list l of type List a with another Coq list L of type List A, where the representation predicate G associated with the elements has type $A \to a \to$ Prop. The definition of Plist is conducted by induction on the list L and by case analysis on the list l. The Coq definition appears next. Observe that the lists l and L must be either both empty or both non-empty.

```
Fixpoint List A a (G:A->a->Hprop) (L:list A) (l:list a) : Hprop :=
  match L,l with
  | nil  , nil   => [True]
  | X::L', x::l' => (x ~> G X) \* (l' ~> List G L')
  | _    , _     => [False]
  end.
```

An example of use of the predicate Plist is given later on.

3.2.4 Focus operations for lists

In this section, I describe focus and unfocus lemmas for mutable lists. I first present the lemmas that are used in practice, and explain afterwards how those lemmas can be derived as immediate corollaries of a few lower-level lemmas. Note that the contents of this section could be directly applied to other linear data structures such as mutable trees and purely-functional lists. Note also that the lemmas presented in this section have to be stated and proved manually. Indeed, the current version of CFML is only able to generate focus and unfocus lemmas for data structures that do not involve null pointers.

I start with the unfocus operation for empty lists. There are two versions. The first one asserts that if a pointer l has for model the empty list then the pointer l must be the null pointer. The second one reciprocally asserts that if a null pointer has for model a list L then the list L must be the empty list.

```
Lemma unfocus_nil : forall a A (G:A->a->Hprop) (l:Loc),
  l ~> Mlist G nil ==> [l = null].

Lemma unfocus_nil' : forall a A (G:A->a->Hprop) (L:list A),
  null ~> Mlist G L ==> [L = nil].
```

The focus operation for the empty list asserts that a null pointer has for model the empty heap. This statement does not involve any pre-condition, so the left-hand side is the empty heap predicate.

```
Lemma focus_nil : forall a A (G:A->a->Hprop),
  [] ==> null ~> Mlist G nil.
```

There are two focus operations for non-empty lists. The first one asserts that if a pointer l has for model the list $X :: L'$, then this pointer points to a reference

cell that contains a pair of a value x modelled as X and of a pointer l' modelled as L'. The values x and l' are existentially quantified. Henceforth, for the sake of readability, I do not show the universal quantifications at the head of statements.

```
Lemma focus_cons :
  (l ~> Mlist G (X::L')) ==>
  Hexists x l', (l ~~> (x,l')) \* (x ~> G X) \* (l' ~> Mlist G L').
```

The second focus operation asserts that if a non-null pointer l has for model the list L, then this list decomposes as $X :: L'$, and l points to a reference cell that contains a pair of a value x modelled as X and of a pointer l' modelled as L'.

```
Lemma focus_cons' :
  [l <> null] \* (l ~> Mlist G L) ==> Hexists x l', Hexists X L',
    [L = X::L'] \* (l ~~> (x,l')) \* (x ~> G X) \* (l' ~> Mlist G L').
```

The unfocus lemma for non-empty lists states the reciprocal entailment of the focus operations.

```
Lemma unfocus_cons :
  (l ~~> (x,l')) \* (x ~> G X) \* (l' ~> Mlist G L') ==>
  (l ~> Mlist G (X::L')).
```

All the lemmas presented in this section can be derived from three core lemmas. Those lemmas are stated with an equality symbol, which corresponds to logical equivalence between heap predicates (recall that I work in Coq with predicate extensionality). In other words, the equality $H_1 = H_2$ holds when both H_1 entails H_2 and H_2 entails H_1. Working with an equality reduces the number of times the definition of Mlist needs to be unfolded in proofs.

```
Lemma models_iff :
  (l ~> Mlist G L)  =  [l = null <-> L = nil] \* (l ~> Mlist G L).
Lemma models_nil :
  []  =  (null ~> Mlist G nil).
Lemma models_cons :
  l ~> Mlist G (X::L')  =  Hexists x l',
    (l ~~> (x,l')) \* (x ~> G X) \* (l' ~> Mlist G L').
```

3.2.5 Example: length of a mutable list

The function length computes the length of a mutable list, using a while loop to traverse the list. It involves a counter n for counting the items that have already been passed by, as well as a pointer h for maintaining the current position in the list.

```
let length (l : 'a mlist) : int =
   let n = ref 0 in
```

```
let h = ref l in
while (!h) != null do
  incr n;
  h := tail !h;
done;
!n
```

The specification of the length function is stated in terms of the representation predicate Mlist. If the pointer l given as input to the function describes a mutable list modelled by the Coq list L, then the function returns an integer equal to the length of L. The input list is not modified, so the post-condition also mentions a copy of the pre-condition. Note that the references n and h allocated during the execution of the function are local; they do not appear in the specification.

```
Lemma length_spec : forall a,
  Spec length (l:Loc) |R>> forall A (G:A->a->Hprop) L
    R (l ~> Mlist G L) (\= length L \*+ l ~> Mlist G L).
```

I explain later how to prove this specification correct without involving a loop invariant. In this section, I follow the standard proof technique, which is based on a loop invariant and involves the manipulation of segments of mutable lists. The representation predicate MlistSeg $e\,G$ describes a list segment that ends on a pointer e. It relates a pointer l (the head of the list segment) with a Coq list L if the items contained between the pointer l and the pointer e are modelled by the list L. So, a list segment heap predicate takes the form $l \rightsquigarrow \mathsf{MlistSeg}\,e\,G\,L$. The definition of MlistSeg directly extends that of Mlist.

```
Fixpoint MlistSeg (e:Loc) A a (G:A->a->Hprop) (L:list A) (l:Loc) :=
  match L with
  | nil => [l = e]
  | X::L' => l ~> Ref (Pair G (Mlist G)) (X,L')
  end.
```

A mutable list is a segment of mutable list that ends on the null pointer. Hence, Mlist is equivalent to MlistSeg null. In fact, I use this result as a definition for Mlist.

```
Definition Mlist := MlistSeg null.
```

The loop invariant for the while-loop involved in the length function decomposes the list in two parts: the segment of list made of the cells that have already been passed by, and the list made of the cells that remain ahead. So, the mathematical model L is decomposed in two sub-lists L_1 and L_2. If e is the pointer contained in the reference h, then the list L_1 models the segment that starts at the pointer l and ends at the pointer e, and the list L_2 models the list that starts at pointer e. The counter n is equal to the length of L_1. During the execution of the loop, the pointer h traverses the list, so the length of L_1 increases and the length of L_2 decreases. I use the list L_2 as index for the loop invariant (recall §3.1.4). The loop invariant is then defined as follows.

```
Definition inv L2 := Hexists e L1,
  [L = L1++L2] \* (n ~~> length L1) \* (h ~~> e)
  \* (l ~> MlistSeg e G L1) \* (e ~> MList G L2).
```

One can check that the instantiation of L_2 as the list L (in which case L_1 must be the empty list) describes the state before the beginning of the loop, and that the instantiation of L_2 as the empty list (in which case L_1 must be the list L) describes the state after the final iteration of the loop.

At a given iteration, either the pointer e is null, in which case the loop terminates, or the pointer e is not null, in which case the loop body is executed. In this case, the focus lemma for non-empty mutable lists can be exploited (lemma `focus_cons'`) to get the knowledge that the list L_2 takes the form $X :: L_2'$ and that e points to a reference on a pair of the form (x, e') such that X models x and L_2' models the list starting at e'. An unfocus operation that operates at the tail of a list segment can be used to extend the segment ranging from l to e into a segment ranging from l to e' (the statement of this unfocus lemma is not shown here). The other facts involved in checking that the execution of the loop body leaves a heap satisfying the invariant instantiated with the list L_2' are straightforward to prove.

3.2.6 Aliased data structures

The heap predicate $l \rightsquigarrow \mathsf{Mlist}\,(\mathsf{Ref}\,\mathsf{Id}_{\mathsf{Int}})\,L$ describes a mutable list that contains distinct references as elements. Some algorithms, however, involve manipulating a mutable list containing possibly-aliased pointers, in the sense that two pointers from the list may point to a same reference. In this case, one would use the predicate $l \rightsquigarrow \mathsf{Mlist}\,\mathsf{Id}_{\mathsf{Loc}}\,L$ instead, in which L is a list of locations (of type $\mathsf{List}\,\mathsf{Loc}$). To describe the references being pointed to by pointers from the list L, we need a new heap predicate that describes a set of mutable data structures.

A value of type Heap describes a set of memory cells. It is isomorphic to a map from locations to values. Values of type Heap only offer a low-level view on memory, because they do not make use of representation predicates for describing the elements contained in the memory cells. So, I introduce a "group predicate", written Group, to describe a piece of heap containing a set of mutable data structures of the same type. If T is a representation predicate of type $A \to \mathsf{Loc} \to \mathsf{Prop}$, and if M is a map from locations to values of type A, then the predicate $\mathsf{Group}\,G\,M$ describes a disjoint union of heaps of the form $x_i \rightsquigarrow G\,X_i$, where x_i denotes a key from the map M and where X_i denotes the value bound to x_i in M. The group predicate is defined using a fold operation on the map M to iterate applications of the separating conjunction, as follows.

```
Definition Group A (G:A->loc->hprop) (M:map Loc A) : Hprop :=
  Map.fold (fun (acc:Hprop) (x:Loc) (X:A) => acc \* (x ~> G X))
           [] M.
```

The focus operation consists in taking one element out of a group. Reciprocally, the unfocus operation on a group consists in adding one element to a group. The

formal statement of those operations involves manipulation of finite maps. Insertion or update of a value in a map is written `M\[x:=X]`, reading is written `M\[x]`, and removal of a binding is written `M\|x`. Moreover, the proposition `index M x` asserts that `x` is a key bound in `M` (reading at a key that is not bound in a map returns an unspecified value). The statements of the focus and unfocus operations on group are then expressed as follows.

```
Lemma group_unfocus :
  forall A (G:A->loc->hprop) (M:map Loc A) (x:Loc) (X:A),
  (Group G M) \* (x ~> G X) ==> (Group G (M\[x:=X]))

Lemma group_focus :
  forall A (G:A->loc->hprop) (M:map Loc A) (x:Loc), index M x ->
  (Group G M) ==> (Group G (M\|x)) \* (x ~> G (M\[x]))
```

Most often, group predicates are not manipulated directly but through derived specifications. For example, consider a mutable list of possibly-aliased integer references. The corresponding heap is described by a mutable list modelled as L, by a group predicate modelled as M, and by a proposition asserting that values from L are keys in M.

$$l \rightsquigarrow \mathsf{Mlist}\,\mathsf{Id}_{\mathsf{Loc}}\,L \;\;*\;\; \mathsf{Group}\,(\mathsf{Ref}\,\mathsf{Id}_{\mathsf{Int}})\,M \;\;*\;\; [\forall x.\, x \in L \Rightarrow x \in \mathrm{dom}(M)]$$

To read at a pointer occurring in the list L, one can use a derived specification for `get` specialized for reading in groups of the form $\mathsf{Group}\,(\mathsf{Ref}\,\mathsf{Id}_A)\,M$. This specification asserts that reading at a location x that belongs to a group described by M returns the value `M[x]`.

```
Spec get (x:Loc) |R>> forall A (M:map Loc A), index M x ->
   R (Group (Ref Id) M) (\= M\[x] \*+ Group (Ref Id) M).
```

3.2.7 Example: the swap function

A standard example for illustrating the reasoning on possibly-aliased pointers is the swap function, which permutes the contents of two references. In the case where the two pointers r_1 and r_2 given as argument to the swap function are distinct, the following specification applies.

```
Spec swap (r1:Loc) (r2:Loc) |R>> forall G1 G2 X1 X2,
  R ((r1 ~> Ref G1 X1) \* (r2 ~> Ref G2 X2))
  (# (r2 ~> Ref G1 X1) \* (r1 ~> Ref G2 X2)).
```

This specification requires that r_1 and r_2 be two distinct pointers, otherwise it is not possible to realize the pre-condition. Nevertheless, the code of the function swap works correctly even when the two pointers r_1 and r_2 are equal. The swap function admits another specification covering this case.

```
Spec swap (r1:Loc) (r2:Loc) |R>> forall G X, r1 = r2 ->
  R ((r1 ~> G X) (# (r1 ~> G X)).
```

It is straightforward to establish both specifications independently. For the swap function, which is very short, it suffices to verify twice the same piece of code. However, in general, we need a way to state and prove a single specifications that handles both the case of non-aliased arguments and that of aliased arguments.

Such a general specification can be expressed using group predicates. The pre-condition describes a group predicate modelled by a map M that should contain both r_1 and r_2 in its domain. The post-condition describes a group predicate modelled by a map M', which corresponds to a copy of M where the contents of the bindings on r_1 and r_2 have been exchanged.

```
Spec swap (r1:Loc) (r2:Loc) |R>> forall G M,
  index M r1 -> index M r2 ->
  let M' := M \[r1:=M\[r2]] \[r2:=M\[r1]] in
  R (Group G M) (# Group G M').
```

This general specification can be used to derive the earlier two specifications given for the swap function, using the focus and unfocus operations on group predicates.

3.3 Reasoning on loops without loop invariants

Although reasoning on loops using loop invariants is sound, it does not take full advantage of local reasoning. In short, the frame rule can be applied to the entire execution of a loop, however it cannot be applied at a given iteration of the loop in order to frame out pieces of heap from the reasoning on the remaining iterations. In this section, I explain why this problem arises and how to fix it.

After writing this material, I learnt that the limitation of loop invariants in Separation Logic has also been recently pointed out by Tuerk [83]. Both Tuerk's solution and mine are based on the same idea: first generate a reasoning rule that corresponds to the encoding of a while-loop as a recursive function, and then simplify this reasoning rule. This approach is also closely related to Hehner's *specified blocks* [34], where, in particular, loops are described using pre- and post-conditions instead of loop invariants (even though Hehner was not using local reasoning). More detailed comparisons can be found in the related work section (§8.2).

The simplest example exhibiting the failure of loop invariants to take advantage of the frame rule is the function that computes the length of a mutable list. The proof of this function using loop invariants has been presented earlier on (§3.2.5). In this section, I consider a variation of the code where the while-loop is encoded as a local recursive procedure, and I show that this transformation allows us to apply the frame rule and to simplify the proof significantly, in particular by removing the need to involve list segments.

3.3.1 Recursive implementation of the length function

The code of the length function, where the while loop has been replaced by a local recursive procedure called `aux`, appears next.

```
let length (l : 'a mlist) =
   let n = ref 0 in
   let h = ref l in
   let rec aux () =
      if (!h) != null then begin
         incr n;
         h := tail !h;
         aux()
      end in
   aux ();
   !n
```

The function `aux` admits a pre-condition that describes a reference n containing a value k and a reference h containing a location e such that e points to a mutable list modelled by L_2. Its post-condition asserts that the reference n contains the value k augmented by the length of the list L_2, that the reference h contains the null pointer, and that e still points to a mutable list modelled by L_2. Note: n and h are free variables of the specification of `aux` shown next.

```
Spec aux () |R>> forall k e L2,
  R ((n ~~> k) \* (h ~~> e) \* (e ~> Mlist G L2))
    (# (n ~~> k + length L2) \* (h ~~> null) \* (e ~> Mlist G L2)).
```

Observe that, contrary to the loop invariant from the previous section, the specification involved here does not refer to the list segment L_1 made of the cells that have already been passed by. The absence of reference to L_1 is made possible by the fact that the frame rule can then be applied around the recursive call to the function `aux`. Intuitively, every time a list cell is being passed by, it is framed out of the reasoning.

Let me explain in more details how the frame rule is applied. Assume that the location e contained in the reference h is not the null pointer. The list L_2 can be decomposed as $X :: L_2'$, and e points towards a pair of the form (x, e'), where X models x and L_2' models x. Just before making the recursive call to the function `aux`, the heap satisfies the predicate shown next.

```
    (n ~~> k+1) \* (h ~~> e) \* (e' ~> Mlist G L2')
\* (x ~> G X) \* (e ~~> (x,e'))
```

The frame rule can be applied to exclude the two heap predicates that are mentioned in the second line of the above statement. The specification of the function `aux` can then be exploited on the predicates from the first line. Combining the post-condition of the recursive call to `aux` with the predicates that have been framed out, we obtain a predicate describing the heap that remains after the execution of `aux`.

```
    (n ~~> k+1+length L2') \* (h ~~> null) \* (e' ~> Mlist G L2')
\* (x ~> G X) \* (e ~~> (x,e'))
```

It remains to verify that the above predicate entails the post-condition of `aux`. To that end, it suffices to apply the unfocus lemma to undo the focus operation performed earlier on, and to check that the length of L_2 is equal to one plus the length of L_2' (recall that L_2' is the tail of L_2).

Encoding the while-loop into a recursive function has led to a dramatic simplification of the proof of correctness. The gain would be even more significant in an algorithm that traverses a mutable tree, as the possibility to apply the frame rule typically saves the need to carry the description of a tree structure with exactly one hole inside it (i.e., the generalization of a list segment to a tree structure).

3.3.2 Improved characteristic formulae for while-loops

I now explain how to produce characteristic formulae for while-loops that directly support local reasoning.

The term "$\mathsf{while}\, t_1\, \mathsf{do}\, t_2$" and the term "$\mathsf{if}\, t_1\, \mathsf{then}\, (t_2\,;\, \mathsf{while}\, t_1\, \mathsf{do}\, t_2)\, \mathsf{else}\, tt$" admit exactly the same semantics (recall that tt denotes the unit value). So, the characteristic formulae of the two terms should be two logically-equivalent predicates, as stated below (recall that bold symbols correspond to notation for characteristic formulae).

$$[\![\mathsf{while}\, t_1\, \mathsf{do}\, t_2]\!] \;=\; \mathbf{if}\; [\![t_1]\!]\; \mathbf{then}\; ([\![t_2]\!]\,;\, [\![\mathsf{while}\, t_1\, \mathsf{do}\, t_2]\!])\; \mathbf{else}\; [\![tt]\!]$$

Let R be a shorthand for the characteristic formula $[\![\mathsf{while}\, t_1\, \mathsf{do}\, t_2]\!]$. The assertion stating that the term "$\mathsf{while}\, t_1\, \mathsf{do}\, t_2$" admits a pre-condition H' and a post-condition Q' is then captured by the proposition $R\,H'\,Q'$. According to the equality stated above, it suffices to prove the proposition $(\mathbf{if}\; [\![t_1]\!]\; \mathbf{then}\; ([\![t_2]\!]\,;R)\; \mathbf{else}\; [\![tt]\!])\,H'\,Q'$ for establishing $R\,H'\,Q'$. For this reason, the characteristic formula for a while loop provides the following assumption.

$$\forall H'Q'.\quad (\mathbf{if}\; [\![t_1]\!]\; \mathbf{then}\; ([\![t_2]\!]\,;R)\; \mathbf{else}\; [\![tt]\!])\,H'\,Q' \quad\Rightarrow\quad R\,H'\,Q'$$

Then, to prove that the evaluation of the term "$\mathsf{while}\, t_1\, \mathsf{do}\, t_2$" admits a particular pre-condition H and a particular post-condition Q, it suffices to establish $R\,H\,Q$ under the above assumption. Notice the difference between H and Q, which specify the behavior of all the iterations of the loop, and H' and Q', which can be used to specify any subset of those iterations. In the definition of the characteristic formula for "$\mathsf{while}\, t_1\, \mathsf{do}\, t_2$", the variable R cannot explicitly refer to $[\![\mathsf{while}\, t_1\, \mathsf{do}\, t_2]\!]$, which we are precisely trying to define. Instead, R is quantified universally and the only assumption about it is the fact that it supports application of the frame rule. This assumption is written $\mathsf{is\_local}R$ (the predicate $\mathsf{is\_local}$ is defined further on, in §5.3.3).

$$[\![\mathsf{while}\, t_1\, \mathsf{do}\, t_2]\!] \;\equiv\; \mathsf{local}\,(\lambda HQ.\;\forall R.\;\mathsf{is\_local}R \wedge \mathcal{H} \Rightarrow R\,H\,Q)$$
$$\text{where}\quad \mathcal{H} \;\equiv\; \forall H'Q'.\,(\mathbf{if}\; [\![t_1]\!]\; \mathbf{then}\; ([\![t_2]\!]\,;R)\; \mathbf{else}\; [\![tt]\!])\,H'\,Q' \Rightarrow R\,H'\,Q'$$

Remark: the above characteristic formula is stronger than the one stated using a loop invariant and a well-founded relation (this result is proved in Coq and exploited by tactics that support reasoning on loops using loop invariants).

Let me illustrate the application of such a characteristic formula on the example of the while-loop implementation of the length function. When reasoning on the loop, a variable R is introduced from the characteristic formula and the goal is as follows.

```
R ((n ~~> 0) \* (h ~~> l) \* (l ~> Mlist G L))
  ((n ~~> length L) \* (h ~~> null) \* (l ~> Mlist G L))
```

This statement is first generalized for the induction, by universally quantifying over the list L, over the pointer l, and over the initial contents k of the counter n.

```
forall L l k,
R ((n ~~> k) \* (h ~~> l) \* (l ~> Mlist G L))
  ((n ~~> k + length L) \* (h ~~> null) \* (l ~> Mlist G L))
```

The proof is then conducted by induction on the structure of L. In the body of the loop, in the particular case where the pointer l is not null, the counter n is incremented and the rest of the loop is executed to compute the length of the tail of the list. Let l' denote the pointer on the tail of the list, and let $X :: L'$ be the decomposition of L. The proof obligation associated with the verification of the remaining executions of the loop body is shown next.

```
R ((n ~~> k+1) \* (h ~~> l') \* (l' ~> Mlist G L')
                   \* (l ~~> (x,l')) \* (x ~> G X))
  ((n ~~> k+1+length L') \* (h ~~> null) \* (l' ~> Mlist G L')
                   \* (l ~~> (x,l')) \* (x ~> G X))
```

At this point, the heap predicates from the second and the fourth line can be framed out, and the remaining statement follows directly from the induction hypothesis.

3.3.3 Improved characteristic formulae for for-loops

I construct a local-reasoning version of the characteristic formula for for-loops in a very similar manner as for while-loops. In the case of a while-loop, I quantified over some predicate R, where $R\,H\,Q$ meant "under the pre-condition H, the loop terminates and admits the post-condition Q". Contrary to while-loops, for-loops involve a counter whose value changes at every iteration. So, to handle a for-loop, I quantify instead over a predicate, called S, that depends on the initial value of the counter. The proposition $S\,a\,H\,Q$ means that the execution of the term "$\mathsf{for}\,i = a\,\mathsf{to}\,b\,\mathsf{do}\,t$" admits the pre-condition H and the post-condition Q.

The characteristic formula for for-loops is then stated as shown next. The main goal is $S\,a\,H\,Q$, and the main assumption relates the predicate $S\,i$, which describes the behavior of the loop starting from index i, with $S\,(i+1)$, which describes the

behavior of the loop starting from index $i+1$. The assumption $\mathsf{is\_local}_1\, S$ asserts that the predicate $S\,i$ supports the frame rule for any index i.

$$[\![\mathsf{for}\, i = a\, \mathsf{to}\, b\, \mathsf{do}\, t]\!] \;\equiv\; \mathsf{local}\,(\lambda H Q.\;\; \forall S.\; \mathsf{is\_local}_1\, S \wedge \mathcal{H} \;\Rightarrow\; S\, a\, H\, Q)$$
$$\text{with } \mathcal{H} \;\equiv\; \forall i H' Q'.\; (\mathbf{if}\; i \leq b\; \mathbf{then}\; ([\![t]\!]\,;\, S\,(i+1))\; \mathbf{else}\; [\![\mathit{tt}]\!])\, H'\, Q' \;\Rightarrow\; S\, i\, H'\, Q'$$

3.4 Treatment of first-class functions

In this section, I describe the specification of a counter function, which carries its own local piece of mutable state, and describe the treatment of higher-order functions, with list iterators, with generic combinators such as composition, with functions in continuation-passing style, and with a function manipulating a list of counter functions.

3.4.1 Specification of a counter function

The function make_counter allocates a reference r with the contents 0 and returns a function that, every time it is called, increments the contents of r and then returns the current contents of r.

```
let make_counter () =
  let r = ref 0 in
  (fun () -> incr r; !r)
```

I first describe a naive specification for make_counter that fully exposes the implementation of the local state of the function. This specification asserts that the function creates a reference r with contents 0 and returns a function f specified as follows: for every contents m of the reference r, a call to f returns the value $m+1$ and sets the contents of r to the value $m+1$.

```
Spec make_counter () |R>>
  R [] (fun f => Hexists r, (r ~~> 0) \*
    [Spec f () |R>> forall m,
       R (r ~~> m) (\= (m+1) \*+ r ~~> (m+1)) ]).
```

A better specification can be devised for make_counter, hiding the fact that the counter is implemented using a reference cell. I define the representation predicate Counter, which relates a counter function f with its mathematical model n, where n is the current value of the counter. So, a heap predicate describing a counter f takes the form $f \rightsquigarrow \mathsf{Counter}\, n$. The Coq definition of Counter is shown next.

```
Definition Counter (n:int) (f:Func) : Hprop :=
  Hexists I:int->hprop,
    (I n) \* [Spec f () |R>> forall m,
                R (I m) (\= (m+1) \*+ I (m+1)) ].
```

The post-condition of make_counter simply states that the return value f is a counter whose model is the integer 0.

```
Spec make_counter () |R>>
  R [] (fun f => f ~> Counter 0)
```

So far, reasoning on a call to a counter function still requires the definition of the predicate Counter to be exposed. It is in fact possible to hide the definition of Counter by providing a lemma describing the behaviour of the application of a counter function. This lemma is stated using AppReturns, as shown below.

```
Lemma Counter_apply : forall f n,
  AppReturns f tt (f ~> Counter n)
                  (\= (n+1) \*+ f ~> Counter (n+1)).
```

Note: the packing of local state in the representation predicate Counter is similar to Pierce and Turner's encoding of objects [73].

To summarize, a library implementing a counter function only needs to expose three things: an abstract predicate Counter, the lemma Counter_apply, and the specification of the function make_counter stated in terms of the abstract predicate Counter. This example illustrates how local state can be entirely packed into representation predicates and need not be mentioned in specifications.

3.4.2 Generic function combinators

In this section, I describe the specification of the combinators apply and compose. Their definition is recalled next.

```
let apply f x = f x
let compose g f x = g (f x)
```

The specification of apply states that, for any pre-condition H and for any post-condition Q, if the application of f to the argument x admits H as pre-condition and Q as post-condition (written AppReturns $f\,x\,H\,Q$), then the application of apply to the arguments f and x admits the pre-condition H and the post-condition Q (written $R\,H\,Q$ below).

```
Spec apply (f:Func) (x:A) |R>> forall (H:Hprop) (Q:B->hprop),
  AppReturns f x H Q -> R H Q.
```

The fact that "AppReturns $f\,x\,H\,Q$" is a sufficient condition for proving the proposition "$[\![\mathsf{apply}\,f\,x]\!]\,H\,Q$" should not be surprising. Indeed, "AppReturns $f\,x$" is the characteristic formula of "$f\,x$" and "$[\![\mathsf{apply}\,f\,x]\!]\,H\,Q$" is the characteristic formula of "apply $f\,x$", and "apply $f\,x$" has the same semantics as "$f\,x$".

The quantification over a pre- and a post-condition in a specification is a typical pattern when reasoning on functions with unknown specifications. This technique is applied throughout this section, and also in the next one which is concerned with reasoning on functions in continuation-passing style.

The specification of compose is shown next. It quantifies over an initial pre-condition H, over a final post-condition Q, and also over an intermediate post-condition Q', which describes the output of the application of the argument f to the argument x.

```
Spec compose (g:Func) (f:Func) (x:A) |R>> forall H Q Q',
  AppReturns f x H Q' ->
  (forall y, AppReturns g y (Q' y) Q) ->
  R H Q.
```

The hypotheses closely resemble the premises of the reasoning rule for let-bindings. Indeed, the term "compose g f x" has the same semantics as the term "let $y = f\,x$ in $g\,y$". In fact, we can use the notation for characteristic formulae to give a more concise specification to the function compose, as shown below. The bold keywords associated with notation for characteristic formulae are written in Coq using capitalized keywords. In particular, App is a notation for the predicate AppReturns.

```
Spec compose (g:Func) (f:Func) (x:A) |R>> forall H Q,
  (Let y = (App f x) In (App g y)) H Q -> R H Q.
```

With this specification, reasoning on an application of the function compose produces exactly the same proof obligation as if the source code of the function compose was inlined. In general, code inlining is not a satisfying approach to verifying higher-order functions because it is not modular. However, the source code of the function compose is so basic that its specification is not more abstract that its source code.

3.4.3 Functions in continuation passing-style

The CPS-append function has been proposed as a verification challenge by Reynolds [77]. This function takes three arguments: two lists x and y and a continuation k. Its purpose is to call the continuation k on the list obtained by concatenation of the lists x and y. The function is implemented recursively, so as to compute the concatenation of x and y using the function itself. I study first the purely-functional version of CPS-append, and then the corresponding imperative version that uses mutable lists.

The source code of the pure CPS-append function is shown next.

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

An example where x is instantiated as a singleton list of the form "v :: nil" helps understand the working of the function.

$$\mathsf{cps\_app}\,(v :: \mathsf{nil})\,y\,k = \mathsf{cps\_app}\,\mathsf{nil}\,y\,(\lambda z.\,k(v :: z)) = (\lambda z.\,k(v :: z))\,y = k(v :: y)$$

The specification of the function asserts that if the application of the continuation k to the concatenation of x and y admits a pre-condition H and a post-condition Q, then so does the application of the CPS-append function to the arguments x, y and k. (For the sake of simplicity, I here assume the lists to contain purely-functional values, thereby avoiding the need for representation predicates.)

```
Spec cps_app (x:list a) (y:list a) (k:Func) |R>>
  forall H Q,  AppReturns k (x++y) H Q  ->  R H Q.
```

This specification is proved by induction on the structure of the list x.

The imperative version of the CPS-append function is shown next. The function tail and set_tail are used to obtain and to update the tail of a mutable list, respectively.

```
let rec cps_app' (x:'a mlist) (y:'a mlist) (k:'a mlist->'b) : 'b =
  if x == null
    then k y
    else cps_app' (tail x) y (fun z -> set_tail x z; k x)
```

The specification is slightly more involved than in the purely-functional case because it involves representation predicates for the list and for the elements of that list. The pre-condition asserts that the pointers x and y are starting points of lists modelled as L and M, respectively. It also mentions a heap predicate H covering the rest of the heap. We need H because the frame rule does not apply when reasoning with CPS functions, as the entire heap usually needs to be passed on to the continuation. In the imperative CPS-append function, the continuation k is called on a pointer z that points to a list modelled as the concatenation of L and M. In addition to the representation predicate associated with z, the pre-condition of k also mention the heap predicate H. The post-condition of the function, called Q, is then the same as the post-condition of the continuation. The complete specification appears next.

```
Spec cps_app' (x:Loc) (y:Loc) (k:Func) |R>>
  forall (G:a->A->hprop) (L M:list A) H Q,
  (forall z, AppReturns k z (z ~> Mlist G (L++M) \* H) Q) ->
  R (x ~> Mlist G L \* y ~> Mlist G M \* H) Q.
```

This specification is proved by induction on the structure of the list L.

3.4.4 Reasoning about list iterators

In this section, I show how to specify the iter function on purely-functional lists, and illustrate the use of this specification for reasoning on the manipulation of a list of counter functions.[1]

[1]Due to lack of time, I do not describe the specification of the functions fold and map, nor the specification of iterators on imperative maps. I will add those later on.

The specification of the function iter shares a number of similarities with the treatment of loops. Again, I want to avoid the introduction of a loop invariant and use a presentation that takes advantage of local reasoning, because I want to be able to reason about the application of the function iter to a function f and to a list l without asserting that some prefix of the list l has been treated by the function f while the remaining segment of the list l has not yet been treated.

Recall the recursive implementation of the function iter.

```
let rec iter f l = match l with
   | [] -> ()
   | x::l -> f x; iter f l
```

The function iter is quite similar to the recursive encoding of a for-loop, in the sense that the argument changes at every recursive call. So, I quantify over a predicate, called S, that depends on the argument l. The proposition $S\,l\,H\,Q$ asserts that the application of "iter f" to the list l admits the pre-condition H and the post-condition Q.

In the specification shown below, l_0 denotes the initial list passed to iter, and l denotes a sublist of l_0. The term "iter $f\,l_0$" admits H and Q as pre- and post-conditions, written $R\,H\,Q$ (where R is bound by the Spec notation), if the proposition $S\,l_0\,H\,Q$ does hold. The hypothesis given about S is a slightly simplified version of the characteristic formula associated with the body of the function iter.

```
Spec iter (f:val) (l0:list a) |R>>
  (forall (S:(list a)->hprop->(unit->hprop)->Prop),
    is_local_1 S ->
    (forall l H' Q',
       match l with
       | nil => (H' ==> Q' tt)
       | x::l' => (App f x ;; S l') H' Q'
       end ->
     S l H' Q') ->
   S l0 H Q) ->
 R H Q.
```

To check the usability of the specification of the function iter, I consider an example. The function step_all, whose code appears below, takes as argument a list of distinct counter functions (recall §3.4.1), and makes a call to each of those counters for incrementing their local state. To keep the example simple, the results produced by the invocation of the counter functions are simply ignored.

```
let step_all (l:list(unit->int)) =
   List.iter (fun f => ignore (f())) l
```

A list of counters is modelled using the application of the representation predicate Plist to the representation predicate Counter. More precisely, if l is a list of functions

and L is a list of integers, then the predicate $l \rightsquigarrow \mathsf{Plist\ Counter}\, L$ asserts that L contains the integer values describing the current state of each of the counters from the list l. A call to the function step_all on the list l increments the state of every counter, so the model of l evolves from the list L to a list L'. The list L' is obtained by applying the successor function to the elements of L, written $\mathsf{map}\,(\lambda n.\, n+1)\, L$, where map denotes the map function on Coq lists. Hence the specification shown next.

```
Spec step_all (l:list Func) |R>> forall (L:list int),
  R (l ~> List Counter L)
    (# l ~> List Counter (map (fun n => n+1) L)).
```

Chapter 4

Characteristic formulae for pure programs

This chapter describes the construction, pretty-printing and manipulation of characteristic formulae for purely-functional programs. I start with the presentation of a set of informal rules explaining, for each construct from the source programming language, what logical formula describes it. I then focus on the specification of n-ary functions and formally define the predicate Spec and the notation associated with it. The formal presentation of a characteristic formula generator follows. Compared with the informal presentation, it involves direct support for n-ary functions and n-ary applications, it describes the lifting of program values to the logical level, and it shows how polymorphism is handled. Finally, I explain how to build a set of tactics for reasoning on characteristic formulae.

4.1 Source language and normalization

源语言的语法和语义

Before generating the characteristic formula of a program, I apply a transformation on the source code so as to put the program in a *administrative normal form*. Through this process, the program is 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. Without such a special treatment of basic operators, programs in normal form tend to be hard to read.

先将程序转换为普遍的形式，使其更易读

Normalization维护了语义，并简化了程序的形式推导

The normalization process, which is similar to A-normalization [28], preserves the semantics and greatly simplifies formal reasoning on programs. Moreover, it is straightforward to implement. Similar transformations have appeared in previous work on program verification (e.g., [40, 76]). I omit a formal description of the normalization process because it is relatively standard and because its presentation

is of little interest.

The grammar of terms in administrative normal form includes values, applications of a value to another value, failure, conditionals, let-bindings for terms, and let-bindings for recursive function definitions. The grammar of terms is extended later on with curried n-ary functions and curried n-ary applications (§4.2.4), as well as pattern matching (§4.6.3). The grammar of values includes integers, algebraic values, and function closures. Algebraic data types correspond to an iso-recursive type made of a tagged unions of tuples. In particular, the boolean values true and false are defined using algebraic data type. Note that source code may contain function definitions but no function closure. Function closures, written $\mu f.\lambda x.t$, are only created at runtime, and they are always closed values.

$$
\begin{array}{lcl}
x, f & := & \textit{variables} \\
n & := & \textit{integers} \\
D & := & \textit{algebraic data constructors} \\
v & := & x \;\mid\; n \;\mid\; D(v,\ldots,v) \;\mid\; \mu f.\lambda x.t \\
t & := & v \;\mid\; (v\,v) \;\mid\; \mathsf{crash} \;\mid\; \mathsf{if}\, v\, \mathsf{then}\, t\, \mathsf{else}\, t \;\mid\; \\
 & & \mathsf{let}\, x = t\, \mathsf{in}\, t \;\mid\; \mathsf{let\,rec}\, f = \lambda x.\, t\, \mathsf{in}\, t
\end{array}
$$

In this chapter, I only consider programs that are well-typed in ML with recursive types (I explain in §4.3.1 what kind of recursive types is handled). The grammar of types and type schema is recalled below. Note: in several examples, I use binary pairs and binary sums; those type constructors can be encoded as algebraic data types.

$$
\begin{array}{lcl}
A & := & \textit{type variable} \\
C & := & \textit{algebraic type constructors} \\
\overline{\tau} & := & \textit{lists of types} \\
\tau & := & A \;\mid\; \mathsf{int} \;\mid\; C\,\overline{\tau} \;\mid\; \tau \to \tau \;\mid\; \mu A.\tau \\
\sigma & := & \forall \overline{A}.\tau
\end{array}
$$

The associated typing rules are the standard typing rules of ML; I do not recall them here.

Characteristic formulae are proved sound and complete with respect to the semantics of the programming language. Since characteristic formulae describe the big-step behavior of programs, it is convenient to describe the source language semantics using big-step rules, as opposed to using a set of small-step reduction rules. The big-step reduction judgment, written $t \Downarrow v$, is defined through the standard inductive rules shown in Figure 4.1.

4.2 Characteristic formulae: informal presentation

The key ideas involved in the construction of characteristic formulae are explained next, through an informal presentation. Compared with the formal definitions given afterwards, the informal presentation makes two simplifications: Caml values and Coq values are abusively identified, and typing annotations are omitted.

$$\frac{}{v \Downarrow v} \qquad \frac{([f \to \mu f.\lambda x.t_1]\,[x \to v_2]\,t_1) \Downarrow v}{((\mu f.\lambda x.t_1)\,v_2) \Downarrow v}$$

$$\frac{t_1 \Downarrow v_1 \qquad ([x \to v_1]\,t_2) \Downarrow v}{(\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2) \Downarrow v} \qquad \frac{([f \to \mu f.\lambda x.t_1]\,t_2) \Downarrow v}{(\mathsf{let\,rec}\, f = \lambda x.\, t_1\, \mathsf{in}\, t_2) \Downarrow v}$$

$$\frac{t_1 \Downarrow v}{(\mathsf{if\,true\,then}\, t_1\, \mathsf{else}\, t_2) \Downarrow v} \qquad \frac{t_2 \Downarrow v}{(\mathsf{if\,false\,then}\, t_1\, \mathsf{else}\, t_2) \Downarrow v}$$

Figure 4.1: Semantics of functional programs

$$\begin{array}{lcl}
[\![v]\!] & \equiv & \lambda P.\ (P\ v) \\
[\![f\ v]\!] & \equiv & \lambda P.\ \mathsf{AppReturns}\ f\ v\ P \\
[\![\mathsf{crash}]\!] & \equiv & \lambda P.\ \mathsf{False} \\
[\![\mathsf{if}\ v\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2]\!] & \equiv & \lambda P.\ (v = \mathsf{true} \Rightarrow [\![t_1]\!]\ P) \wedge (v = \mathsf{false} \Rightarrow [\![t_2]\!]\ P) \\
[\![\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2]\!] & \equiv & \lambda P.\ \exists P'.\ [\![t_1]\!]\ P' \wedge (\forall x.\ P'\ x \Rightarrow [\![t_2]\!]\ P) \\
[\![\mathsf{let\,rec}\ f = \lambda x.\, t_1\ \mathsf{in}\ t_2]\!] & \equiv & \lambda P.\ \forall f.\ (\forall x.\, \forall P'.\, [\![t_1]\!]\ P' \Rightarrow \mathsf{AppReturns}\ f\ x\ P') \Rightarrow [\![t_2]\!]\ P
\end{array}$$

Figure 4.2: Informal presentation of the characteristic formula generator

4.2.1 Characteristic formulae for the core language

The characteristic formula of a term t, written $[\![t]\!]$, is generated using an algorithm that follows the structure of t. Recall that, given a post-condition P, the characteristic formula is such that the proposition "$[\![t]\!]\ P$" holds if and only if the term t terminates and returns a value that satisfies P. The definition of $[\![t]\!]$ for a particular term t always takes the form "$\lambda P.\, \mathcal{H}$", where $\mathcal{H}$ expresses what needs to be proved in order to show that the term t returns a value satisfying the post-condition P. The definitions appear in Figure 4.2 and are explained next.

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

$$(v = \mathsf{true} \Rightarrow [\![t_1]\!]\ P) \wedge (v = \mathsf{false} \Rightarrow [\![t_2]\!]\ P)$$

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

defined as "$\lambda P.\ \mathsf{False}$".

To show that a term "$\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2$" returns a value satisfying P, one must prove that there exists a post-condition P' such that t_1 returns a value satisfying P' and that, for any x satisfying P', t_2 returns a value satisfying P. So, the proposition $[\![\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2]\!]\, P$ is equivalent to the proposition "$\exists P'.\ [\![t_1]\!]\, P' \ \wedge\ (\forall x.\ P'\, x \ \Rightarrow\ [\![t_2]\!]\, P)$".

Consider the definition of a possibly-recursive function "$\mathsf{let\,rec}\, f = \lambda x.\, t_1\, \mathsf{in}\, t_2$". The proposition "$\forall x.\, \forall P'.\, [\![t_1]\!]\, P' \Rightarrow \mathsf{AppReturns}\, f\, x\, P'$" is called the *body description* associated with the function definition "$\mathsf{let\,rec}\, f = \lambda x.\, t_1$". It captures the fact that, in order to prove that the application of f to x returns a value satisfying a post-condition P', it suffices to prove that the body t_1, instantiated with that particular argument x, terminates and returns a value satisfying P'. The characteristic formula for a function definition is built upon the body description of that function: to prove that the term "$\mathsf{let\,rec}\, f = \lambda x.\, t_1\, \mathsf{in}\, t_2$" returns a value satisfying P, it suffices to prove that, for any name f, asssuming the body description for "$\mathsf{let\,rec}\, f = \lambda x.\, t_1$" to hold, the term t_2 returns a value satisfying P. Hence, "$[\![\mathsf{let\,rec}\, f = \lambda x.\, t_1\, \mathsf{in}\, t_2]\!]\, P$" is equivalent to:

$$\forall f.\quad (\forall x.\, \forall P'.\, [\![t_1]\!]\, P' \Rightarrow \mathsf{AppReturns}\, f\, x\, P') \ \Rightarrow\ [\![t_2]\!]\, P$$

4.2.2 Definition of the specification predicate

In a language like Caml, functions of several arguments are typically presented in a curried fashion. A definition of the form "$\mathsf{let\,rec}\, f = \lambda x\, y.\, t$" describes a function that, when applied to its first argument x, returns a function that expects an argument y. In order to obtain a realistic tool for the verification of Caml programs, it is crucial to offer direct support for reasoning on the definition and application of n-ary curried functions. In this next section, I give the formal definition of the predicate Spec used to specify unary functions. In the next section I describe the generalization of this predicate to higher arities.

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

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

The same specification can be rewritten with the Spec notation as follows.

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

The notation introduced by the keyword Spec is defined using a family of higher-order predicates called Spec_n, where n corresponds to the arity of the function. The definition of Spec_1 appears next, and its generalization Spec_n for n is given afterwards.

The proposition "$\mathsf{Spec}_1\, f\, K$" asserts that the function f admits the specification K. The predicate K takes both x and R as argument, and specifies the result of the application of f to x. The predicate R is to be applied to the post-condition that

holds of the result of "$f\,x$". For example, the previous specification for half actually stands for:

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

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

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

where K has type $A \to ((B \to \mathsf{Prop}) \to \mathsf{Prop}) \to \mathsf{Prop}$, and where A and B correspond to the input and the output type of f.

For example, consider an application of half to the argument 8. The behavior of this application is described by the proposition "$K\,8\,(\mathsf{AppReturns}_1\,\mathsf{half}\,8)$", where K stands for the specification of the function half, which is "$\lambda x\,R.\ \forall n \geq 0.\ x = 2 * n\ \Rightarrow\ R\,(= n)$". Simplifying the application of K in the expression "$K\,8\,(\mathsf{AppReturns}_1\,\mathsf{half}\,8)$" gives back the first property that was stated about that function:

$$\forall n.\ n \geq 0\ \Rightarrow\ 8 = 2 * n\ \Rightarrow\ \mathsf{AppReturns}_1\,\mathsf{half}\,8\,(= n)$$

The actual definition of Spec_1 includes an extra side-condition, expressing that K is covariant in R. Covariance is formally captured by the predicate Weakenable, defined as follows:

$$\mathsf{Weakenable}\,J \quad \equiv \quad \forall\,P\,P'.\ J\,P \to (\forall x.\,P\,x \to P'\,x) \to J\,P'$$

where J has type "$(B \to \mathsf{Prop}) \to \mathsf{Prop}$" for some type B. The formal definition of Spec_1 appears in the middle of Figure 4.3. Fortunately, thanks to appropriate lemmas and tactics, the predicate Weakenable never needs to be manipulated explicitly while verifying programs.

4.2.3 Specification of curried n-ary functions

Generalizing the definitions of $\mathsf{AppReturns}_1$ and Spec_1 to higher arities is not entirely straightforward, due to the need to support reasoning on partial applications and over applications. Intuitively, the specification of a n-ary curried function f should capture the property that the application of f to less than n arguments terminates and returns a function whose specification is an appropriate specialization of the specification of f.

Firstly, I define the predicate $\mathsf{AppReturns}_n$ in such a way that the proposition "$\mathsf{AppReturns}_n\,f\,v_1 \ldots v_n\,P$" states that the application of f to the n arguments $v_1 \ldots v_n$ returns a value satisfying P. The family of predicates $\mathsf{AppReturns}_n$ is defined by induction on n, ultimately in terms of the abstract predicate AppReturns, as shown at the top of Figure 4.3. For instance, "$\mathsf{AppReturns}_2\,f\,x\,y\,P$" states that the application of f to x returns a function g such that the application of g to y returns a value satisfying P. More generally, if m is smaller than n, then applications at arities n and m are related as follows:

$$\begin{array}{l}\mathsf{AppReturns}_n\,f\,x_1 \ldots x_n\,P \quad \Longleftrightarrow \\ \mathsf{AppReturns}_m\,f\,x_1 \ldots x_m\,(\lambda g.\,\mathsf{AppReturns}_{n-m}\,g\,x_{m+1} \ldots x_n\,P)\end{array}$$

$$\begin{array}{lcl}
\mathsf{AppReturns}_1\, f\, x\, P & \equiv & \mathsf{AppReturns}\, f\, x\, P \\
\mathsf{AppReturns}_n\, f\, x_1 \ldots x_n\, P & \equiv & \mathsf{AppReturns}\, f\, x_1\, (\lambda g.\ \mathsf{AppReturns}_{n-1}\, g\, x_2 \ldots x_n\, P)
\end{array}$$

$$\begin{array}{lcl}
\mathsf{is_spec}_1\, K & \equiv & \forall x.\, \mathsf{Weakenable}\, (K\, x) \\
\mathsf{is_spec}_n\, K & \equiv & \forall x.\, \mathsf{is_spec}_{n-1}\, (K\, x) \\
\mathsf{Spec}_1\, f\, K & \equiv & \mathsf{is_spec}_1\, K \wedge \forall x.\, K\, x\, (\mathsf{AppReturns}\, f\, x) \\
\mathsf{Spec}_n\, f\, K & \equiv & \mathsf{is_spec}_n\, K \wedge \mathsf{Spec}_1\, f\, (\lambda x\, R.\ R\, (\lambda g.\, \mathsf{Spec}_{n-1}\, g\, (K\, x)))
\end{array}$$

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

Figure 4.3: Formal definitions for $\mathsf{AppReturns}_n$ and Spec_n

Secondly, I define the predicate Spec_n, again by induction on n. For example, a curried function f of two arguments is a total function that, when applied to its first argument x, returns a unary function g that admits a certain specification which depends on that first argument. If K denotes the specification of f then "$K\, x$" denotes the specification of the partial application of f to x. Intuitively, the definition of Spec_2 is as follows.

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

where K has type "$A_1 \to A_2 \to ((B \to \mathsf{Prop}) \to \mathsf{Prop}) \to \mathsf{Prop}$". Note that Spec_2 is polymorphic in the types A_1, A_2 and B. The formal definition of Spec_n, which appears in Figure 4.3, also includes a side condition to ensure that K is covariant in its argument R, written $\mathsf{is\_spec}_n\, K$.

The high-level notation for specifications used in §2.2 can now be easily explained in terms of the family of predicates Spec_n.

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

The predicate Spec_n is ultimately defined in terms of $\mathsf{AppReturns}$. A direct elimination lemma can be proved for relating a specification for a n-ary function stated in terms of Spec_n with the behavior of an application of that function described with the predicate $\mathsf{AppReturns}_n$. This elimination lemma takes the following form.

$$\mathsf{Spec}_n\, f\, K \quad \Rightarrow \quad K\, x_1 \ldots x_n\, (\mathsf{AppReturns}_n\, f\, x_1 \ldots x_n)$$

where f has type Func, where each x_i admits a type A_i, and where K is of type $A_1 \to \ldots A_n \to ((B \to \mathsf{Prop}) \to \mathsf{Prop}) \to \mathsf{Prop}$, with B being the return type of the function f. This elimination lemma is not intended to be used directly; it serves as a basis for proving other lemmas upon which tactics for reasoning on applications are based (§4.7.2). Note that the elimination lemma admits a reciprocal,

which is presented in detail in §4.7.3. This introduction lemma admits the following statement (two side-conditions are omitted).

$$\big(\forall x_1 \ldots x_n.\ K\, x_1 \ldots x_n\, (\mathsf{AppReturns}_n\, f\, x_1 \ldots x_n)\big) \quad \Rightarrow \quad \mathsf{Spec}_n\, f\, K$$

Remark: in the CFML library, the predicates $\mathsf{AppReturns}_n$ and Spec_n are implemented up to $n = 4$. It would not take very long to support arities up to 12, which should be sufficient for most functions used in practice. It might even be possible to define the n-ary predicates in an arity-generic way, however I have not had time to work on such definitions.

4.2.4 Characteristic formulae for curried functions

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

The generation of the characteristic formula for a n-ary application is straightforward, as it directly relies on the predicate $\mathsf{AppReturns}_n$. The definition is:

$$[\![f\, v_1 \ldots v_n]\!] \quad \equiv \quad \lambda P.\ \mathsf{AppReturns}_n\, f\, v_1 \ldots v_n\, P$$

The treatment of n-ary abstractions is slightly more complex. Recall that the body description associated with a function definition "$\mathsf{let\ rec}\ f = \lambda x.\, t$" is the proposition: "$\forall x.\, \forall P.\, [\![t]\!]\, P \Rightarrow \mathsf{AppReturns}\, f\, x\, P$". I could generalize this definition using the predicate $\mathsf{AppReturns}_n$, but this would not capture the fact that partial applications of the n-ary function terminate. Instead, I rely on the predicate Spec_n. In this new presentation, the body description of a n-ary function "$\mathsf{let\ rec}\ f = \lambda x_1 \ldots x_n.\, t$", where $n \geq 1$, is defined as follows.

$$\forall K.\ \mathsf{is\_spec}_n\, K \wedge \big(\forall x_1 \ldots x_n.\ K\, x_1 \ldots x_n\, [\![t]\!]\big) \Rightarrow \mathsf{Spec}_n\, f\, K$$

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

To summarize, the characteristic formula of a n-ary application is expressed in terms of the predicate $\mathsf{AppReturns}_n$ and the characteristic formula of a n-ary function definition is expressed in terms of the predicate Spec_n. The elimination lemma, which relates Spec_n to $\mathsf{AppReturns}_n$, then serves as a basis for reasoning on the application of n-ary functions.

4.3 Typing and translation of types and values

So far, I have abusively identified program values from the programming language with values from the logic. This section clarifies the translation from Caml types to Coq types and the translation from Caml values to Coq values.

4.3.1 Erasure of arrow types and recursive types

I map every Caml type to its corresponding Coq type, except for arrow types. As explained earlier on, due to the mismatch between the programming language arrow type and the logical arrow type, I represent Caml functions using Coq values of type Func.

To simplify the presentation and the proofs about characteristic formulae, it is convenient to consider an intermediate type system, called weak-ML. This type system is like ML except that the arrow type constructor is replaced with a constant type constructor, called func. During the generation of characteristic formulae, variables that have the type func in weak-ML become variables of type Func in Coq. Also, weak-ML does not include general recursive types but only algebraic data types, such as the type of lists. The grammar of weak-ML types is thus as follows.

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

The translation of an ML type τ into the corresponding weak-ML type is written $\langle\tau\rangle$. In short, $\langle\tau\rangle$ is a copy of τ where all the arrow types are replaced with the type func. The erasure operator $\langle\cdot\rangle$ also handles polymorphic types and general recursive types, as explained further on. The formal definition of the erasure operator $\langle\cdot\rangle$ appears in Figure 4.4. Note that $\langle\overline{\tau}\rangle$ denotes the application of the erasure operator to all the elements of the list of types $\overline{\tau}$.

The translation of a Caml type scheme $\forall\overline{A}.\tau$ is a weak-ML type scheme of the form $\forall\overline{B}.\langle\tau\rangle$. The list of types $\overline{B}$ might be strictly smaller than the list $\overline{A}$ because some type variables occurring in τ may no longer occur in $\langle\tau\rangle$. For example, the ML type scheme "$\forall AB.\ A + (B \to B)$" is translated in weak-ML as "$\forall A.\ A + \mathsf{func}$", which no longer involves the type variable B.

An ML program may involve general equi-recursive types, of the form "$\mu A.\tau$". Such recursive types are handled by CFML only if the translation of the type τ no longer refers to the variable A. This is often the case because any occurrence of the variable A under an arrow type gets erased. For example, the term "$\lambda x.\, x\, x$" admits the general recursive type "$\mu A.(A \to B)$". This type is simply translated in weak-ML as func, which no longer involves a recursive type. So, CFML supports reasoning on the value "$\lambda x.\, x\, x$". However, reasoning on a value that admits a type in which the recursivity does not traverse an arrow is not supported. For example, CFML does not support reasoning on a program involving values of type "$\mu A.(A \times \mathsf{int})$".

In summary, CFML supports base types, algebraic data types and arrow types. Moreover it can handle functions that admit an equi-recursive type, because all

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

Figure 4.4: Translation from ML types to weak-ML types

arrow types are mapped to the constant type func.

4.3.2 Typed terms and typed values

The characteristic formula generator takes as input a typed weak-ML program, that is, a program in which all the terms and all the values are annotated with their weak-ML type. In the implementation, the CFML generator takes as input the type-carrying abstract syntax tree produced by the OCaml type-checker, and it applies the erasure operator to all the types carried by that data structure. I explain next the notation employed for referring to typed terms and typed values.

The meta-variable $\hat{t}$ ranges over typed term, $\hat{v}$ ranges over typed values, and $\hat{w}$ ranges over polymorphic typed values. A typed entity is a pair, whose first component is a weak-ML type, and whose second component is a construction from the programming. Typed programs moreover carry explicit information about generalized types variables and type applications. A list of universally-quantified type variables, written "$\Lambda\overline{A}.$", appears in polymorphic let-bindings, in polymorphic values, and in function closures. Note that a polymorphic function is not a polymorphic value because all functions, including polymorphic functions, admit the type func. In the grammar of typed values, explicit type applications are mentioned for variables, written $x\,\overline{T}$, and for polymorphic data constructors, written $D\,\overline{T}$.

$$
\begin{array}{lcl}
\hat{v} & := & x\,\overline{T} \quad | \quad n \quad | \quad D\,\overline{T}(\hat{v},\ldots,\hat{v}) \quad | \quad \mu f.\Lambda\overline{A}.\lambda x.\hat{t} \\
\hat{w} & := & \Lambda\overline{A}.\,\hat{v} \\
\hat{t} & := & \hat{v} \quad | \quad \hat{v}\,\overline{\hat{v}} \quad | \quad \mathsf{crash} \quad | \quad \mathsf{if}\,\hat{v}\,\mathsf{then}\,\hat{t}\,\mathsf{else}\,\hat{t} \\
 & & \mathsf{let}\,x = \Lambda\overline{A}.\,\hat{t}\,\mathsf{in}\,\hat{t} \quad | \quad \mathsf{let\,rec}\,f = \Lambda\overline{A}.\lambda x.\hat{t}\,\mathsf{in}\,\hat{t}
\end{array}
$$

Substitution in typed terms involves polymorphic values. For example, during the evaluation of the term "$\mathsf{let}\,x = \Lambda\overline{A}.\,\hat{t}_1\,\mathsf{in}\,\hat{t}_2$", the term $\hat{t}_1$ reduces to some value $\hat{v}$, and then the variable x is substituted in $\hat{t}_2$ with the polymorphic value "$\Lambda\overline{A}.\,\hat{v}$". Such polymorphic values are written $\hat{w}$.

I write $\hat{t}^T$ to indicate that T is the type that annotates by the typed term $\hat{t}$. A similar convention applies for values, written $\hat{v}^T$. Moreover, I write $\mu f.\Lambda\overline{A}.\lambda x^{T_0}.\hat{t}^{T_1}$

to denote the fact that the input type of the function is T_0 and that its return type is T_1. Finally, given a typed term $\hat{t}$, one may strip all the types and obtained a corresponding raw term t, which is written "$\mathsf{strip\_types}\ \hat{t}$".

Remark: to simplify the soundness proof, I enforce the invariant that bound polymorphic variables must occur at least once in the type on which they scope. In particular, in a function closure of the form $\mu f.\Lambda\overline{A}.\lambda x^{T_0}.\hat{t}^{T_1}$, a type variable from the list $\overline{A}$ should occur in the type T_0 or in type T_1 (or in both).[1]

4.3.3 Typing rules of weak-ML

A typed term $\hat{t}$ denotes a term in which every node is annotated with a weak-ML type. However, it is not guaranteed that those type annotations are coherent throughout the term. Since the construction of characteristic formulae only makes sense when type annotations are coherent, I introduce a typing judgment for weak-ML typed terms, written $\Delta \vdash \hat{t}$, or simply $\vdash \hat{t}$ when the typing context Δ is empty.

The proposition $\Delta \vdash \hat{t}$ captures the fact that the type-annotated term $\hat{t}$ is well-typed in a context Δ, where Δ maps variables to weak-ML type schema. The typing rules defining this typing judgment appear in Figure 4.5. In those typing rules, typed terms are systematically annotated with the type they carry. Note that the typing rule for data constructors involves a premise describing the type of a data constructor: the proposition "$D : \forall\overline{A}.\ (T_1 \times \ldots \times T_n) \to C\,\overline{A}$" asserts that the data constructor D is polymorphic in the types $\overline{A}$, and that it expects arguments of type T_i and constructs a value of type $C\,\overline{A}$.

Compared with the typing rules that one would have for terms annotated with ML types, the only difference is that arrow types are replaced with the type func. This results in the typing rule for applications, shown in the upper-left corner of the figure, being totally unconstrained: a function of type func can be applied to an argument of some arbitrary type T_1, and pretend to produce a result of some arbitrary type T_2.

Throughout the rest of this chapter, all the typed values and typed terms being manipulated are well-typed in weak-ML.

4.3.4 Reflection of types in the logic

In this section, I define the translation from weak-ML types to Coq types. This translation is almost the identity, since every type constructor from weak-ML is directly mapped to the corresponding Coq type constructor. Yet, it would be confusing to identify weak-ML types with Coq types. So, I introduce an explicit reflection operator: $[\![T]\!]$ denotes the Coq type that corresponds to the weak-ML type T. The

[1] If the invariant is not satisfied, then one can replace the dangling free type variables with an arbitrary type, say int, without changing the semantics of the term nor its generality. For example, the function $\mu f.\Lambda A.\lambda x^{\mathsf{int}}.((\lambda y^{(\mathsf{list}\ A)}.\ x)\ \mathsf{nil})^{\mathsf{int}}$ has type "$\forall A.\ \mathsf{int} \to \mathsf{int}$". The type variable A is used to type-check the empty list nil which plays no role in the function. The typing of the function can be updated to $\mu f.\Lambda.\lambda x^{\mathsf{int}}.((\lambda y^{(\mathsf{list}\ \mathsf{int})}.\ x)\ \mathsf{nil})^{\mathsf{int}}$, which no longer involves a dangling type variable.

$$\frac{\Delta \vdash \hat{f}^{\mathsf{func}} \qquad \Delta \vdash \hat{v}^{T_1}}{\Delta \vdash (\hat{f}^{\mathsf{func}}\ \hat{v}^{T_1})^{T_2}} \qquad \frac{\Delta \vdash \hat{v}^{\mathsf{bool}} \qquad \Delta \vdash \hat{t}_1{}^{T} \qquad \Delta \vdash \hat{t}_2{}^{T}}{\Delta \vdash (\mathsf{if}\ \hat{v}^{\mathsf{bool}}\ \mathsf{then}\ \hat{t}_1{}^{T}\ \mathsf{else}\ \hat{t}_2{}^{T})^{T}} \qquad \frac{}{\Delta \vdash \mathsf{crash}^{T}}$$

$$\frac{\Delta, \overline{A} \vdash \hat{t}_1{}^{T_1} \qquad \Delta, x : \forall \overline{A}.T_1 \vdash \hat{t}_2{}^{T_2}}{\Delta \vdash (\mathsf{let}\ x = \Lambda \overline{A}.\ \hat{t}_1{}^{T_1}\ \mathsf{in}\ \hat{t}_2{}^{T_2})^{T_2}}$$

$$\frac{\Delta, \overline{A}, f : \mathsf{func}, x : T_0 \vdash \hat{t}_1{}^{T_1} \qquad \Delta, f : \mathsf{func} \vdash \hat{t}_2{}^{T_2}}{\Delta \vdash (\mathsf{let\,rec}\ f = \Lambda \overline{A}.\lambda x^{T_0}.\hat{t}_1{}^{T_1}\ \mathsf{in}\ \hat{t}_2{}^{T_2})^{T_2}} \qquad \frac{(x : \forall \overline{A}.T) \in \Delta}{\Delta \vdash (x\,\overline{T})^{([\overline{A} \to \overline{T}]\,T)}}$$

$$\frac{}{\Delta \vdash n^{\mathsf{int}}} \qquad \frac{D : \forall \overline{A}.\ (T_1 \times \ldots \times T_n) \to C\,\overline{A} \qquad \forall i.\quad \Delta \vdash \hat{v}_i{}^{([\overline{A} \to \overline{T}]\,T_i)}}{\Delta \vdash (D\,\overline{T}(\hat{v}_1, \ldots, \hat{v}_n))^{(C\,\overline{T})}}$$

$$\frac{\Delta, \overline{A}, f : \mathsf{func}, x : T_0 \vdash \hat{t}_1{}^{T_1}}{\Delta \vdash (\mu f.\Lambda \overline{A}.\lambda x^{T_0}.\hat{t}_1{}^{T_1})^{\mathsf{func}}} \qquad \frac{\Delta, \overline{A} \vdash \hat{v}^{T}}{\Delta \vdash (\Lambda \overline{A}.\ \hat{v}^{T})^{(\forall \overline{A}.T)}}$$

Figure 4.5: Typing rules for typed weak-ML terms

formal definition of the reflection operator $\lceil\!\lceil \cdot \rceil\!\rceil$ appears below.

$$\begin{array}{lcl} \lceil\!\lceil A \rceil\!\rceil & \equiv & A \\ \lceil\!\lceil \mathsf{int} \rceil\!\rceil & \equiv & \mathsf{Int} \\ \lceil\!\lceil C\,\overline{T} \rceil\!\rceil & \equiv & C\,\lceil\!\lceil \overline{T} \rceil\!\rceil \\ \lceil\!\lceil \mathsf{func} \rceil\!\rceil & \equiv & \mathsf{Func} \\ \lceil\!\lceil \forall \overline{A}.T \rceil\!\rceil & \equiv & \forall \overline{A}.\ \lceil\!\lceil T \rceil\!\rceil \end{array}$$

Algebraic type definitions are translated into corresponding Coq inductive definitions. This translation is straightforward. For example, Caml lists are translated into Coq lists. Note that the positivity requirement associated with Coq inductive types is not a problem here because all arrow types have been mapped to the type func, so the translation does not produce any negative occurrence of an inductive type in its definition. Thus, for every type constructor C from the source program, a corresponding constant C is defined in Coq. In particular, the translation of the type $C\,\overline{T}$ is written $C\,\lceil\!\lceil \overline{T} \rceil\!\rceil$.

Henceforth, I let $\mathcal{T}$ range over Coq types of the form $\lceil\!\lceil T \rceil\!\rceil$, which are called reflected types. Similarly, I let $\mathcal{S}$ range over Coq types of the form $\lceil\!\lceil S \rceil\!\rceil$, where S is a weak-ML type scheme. Note that a reflected type scheme $\mathcal{S}$ always takes the form $\forall \overline{A}.\mathcal{T}$, for some reflected type $\mathcal{T}$. To summarize, we start with an ML type, written τ, then turn it into a weak-ML type, written T, and finally translate it into a Coq type, written $\mathcal{T}$.

4.3.5 Reflection of values in the logic

I now describe the translation from Caml values to Coq values. The decoding operator, written $\lceil \hat{v} \rceil$, transforms a value $\hat{v}$ of type T into the corresponding Coq value of type $[\![T]\!]$. Since program values might contain free variables, the decoding operator in fact also takes as argument a context Γ that maps Caml variables to Coq variables. The decoding of a value $\hat{v}$ in a context Γ is written $\lceil \hat{v} \rceil^{\Gamma}$. The Coq variables bound by Γ should be typed appropriately: if $\hat{v}$ contains a free variable x of type S then the Coq variable $\Gamma(x)$ should admit the type $[\![S]\!]$.

In the definition of $\lceil v \rceil^{\Gamma}$ shown next, values on the left-hand side are typed weak-ML values and values on the right-hand side are Coq values. Note that Coq values are inherently well-typed, because Coq is based on type theory. The only difficulty is the treatment of polymorphism. When the source program contains a polymorphic variable x applied to some types $\overline{T}$, this occurrence is translated as the application of $\Gamma(x)$ to the translation of the types $\overline{T}$, written $\Gamma(x)\,[\![\overline{T}]\!]$. Similarly, the type arguments of data constructors get translated. Function closures do not exist in source programs, so there is no need to translate them.

$$\begin{array}{lcl}
\lceil x\,\overline{T} \rceil^{\Gamma} & \equiv & \Gamma(x)\,[\![\overline{T}]\!] \\
\lceil n \rceil^{\Gamma} & \equiv & n \\
\lceil D\,\overline{T}(\hat{v}_1, \ldots, \hat{v}_2) \rceil^{\Gamma} & \equiv & D\,[\![\overline{T}]\!]\,(\lceil \hat{v}_1 \rceil^{\Gamma}, \ldots, \lceil \hat{v}_2 \rceil^{\Gamma}) \\
\lceil \mu f.\Lambda\overline{A}.\lambda x.t \rceil^{\Gamma} & \equiv & \textit{not needed at this time}
\end{array}$$

When decoding closed values, the context Γ is typically empty. Henceforth, I write $\lceil \hat{v} \rceil$ as a shorthand for $\lceil \hat{v} \rceil^{\Gamma}$.

The decoding of polymorphic values is not needed for generating characteristic formulae, however it is involved in the proofs of soundness and completeness. Having in mind the definition of the decoding of polymorphic values helps understanding the generation of characteristic formulae for polymorphic let-bindings. The decoding of a closed polymorphic value "$\Lambda\overline{A}.\,\hat{v}$" is a Coq function that expects some types $\overline{A}$ and returns the decoding of the value $\hat{v}$.

$$\lceil \Lambda\overline{A}.\,\hat{v} \rceil \quad \equiv \quad \lambda\overline{A}.\,\lceil \hat{v} \rceil$$

For example, the value $(\mathsf{nil}, \mathsf{nil})$ has type "$\forall A.\,\forall B.\,\mathsf{list}\,A \times \mathsf{list}B$". The Coq translation of this value is "`fun A B => (@nil A, @nil B)`", where the symbol `@` indicates that type arguments are given explicitly.

4.4 Characteristic formulae: formal presentation

This section contains the formal definition of a characteristic formula generator, including the definition of the notation system for characteristic formulae. It starts with a description of the treatment of polymorphism, and it includes the definition of AppReturns.

4.4.1 Characteristic formulae for polymorphic definitions

Consider a polymorphic let-binding "$\mathsf{let}\, x = \Lambda\overline{A}.\, \hat{t}_1\, \mathsf{in}\, \hat{t}_2$", and let T_1 denote the type of $\hat{t}_1$. The variable x has type $\forall\overline{A}.T_1$. I define the characteristic formula associated with that polymorphic let-bindings as follows.

$$\begin{array}{l}[\![\mathsf{let}\, x = \Lambda\overline{A}.\, \hat{t}_1\, \mathsf{in}\, \hat{t}_2]\!] \;\equiv \\ \quad \lambda P.\; \exists P'.\; \big(\forall\overline{A}.\; [\![\hat{t}_1]\!]\; (P'\;\overline{A})\big) \;\wedge\; \big(\forall x.\; (\forall\overline{A}.\; (P'\;\overline{A})\;(x\;\overline{A})) \;\Rightarrow\; [\![\hat{t}_2]\!]\; P\big)\end{array}$$

where P' has type $\forall\overline{A}.([\![T_1]\!] \to \mathsf{Prop})$ and the quantified variable x has type $\forall\overline{A}.[\![T_1]\!]$. The formula first asserts that, for any instantiations of the types $\overline{A}$, the typed term $\hat{t}_1$, which depend on the variables $\overline{A}$, should satisfy the post-condition $(P'\,\overline{A})$. Observe that the post-condition P' describing x is a polymorphic predicate of type $\forall\overline{A}.([\![T_1]\!] \to \mathsf{Prop})$. It is not a predicate of type $(\forall\overline{A}.[\![T]\!]) \to \mathsf{Prop}$, because we only care about specifying monomorphic instances of the polymorphic variable x. A particular monomorphic instance of x in $\hat{t}_2$ takes the form $x\,\overline{T}$. It is expected to satisfy the predicate $P'\,\overline{T}$. Hence the assumption $\forall\overline{A}.\; (P'\;\overline{A})\;(x\;\overline{A})$ provided for reasoning about x in the continuation $\hat{t}_2$.

It remains to see how polymorphism in handled in function definitions. Consider a function definition "$\mathsf{let\,rec}\, f = \Lambda\overline{A}.\lambda x_1 \ldots x_n.\hat{t}_1\, \mathsf{in}\, \hat{t}_2$". The associated body description, shown below, quantifies over the types $\overline{A}$. Those types are involved in the type of the specification K, in the type of the arguments x_i, in the characteristic formula of the body $\hat{t}_1$, and in the implicit arguments of the predicate Spec_n.

$$\forall\overline{A}.\forall K.\; \mathsf{is\_spec}_n\, K \wedge \big(\forall x_1 \ldots x_n.\; K\, x_1 \ldots x_n\, [\![\hat{t}_1]\!]\big) \;\Rightarrow\; \mathsf{Spec}_n\, f\, K$$

4.4.2 Evaluation predicate

The predicate $\mathsf{AppReturns}$ is defined in the CFML library in terms of a lower-level predicate called $\mathsf{AppEval}$, which is one of the three axioms upon which the library is built. The proposition $\mathsf{AppEval}\, F\, V\, V'$ captures the fact that the application of a function whose decoding is F to a value whose decoding is V returns a value whose decoding is V'. The type of $\mathsf{AppEval}$ is as follows.

$$\mathsf{AppEval} \;:\; \forall AB.\; \mathsf{Func} \to A \to B \to \mathsf{Prop}$$

A formal definition that justifies the interpretation of the axiom $\mathsf{AppEval}$ is included in the soundness proof (§6.1.5).

Intuitively, the judgment "$\mathsf{AppReturns}\, F\, V\, P$" asserts that the application of F to the argument V terminates and returns a value V' that satisfies P. The predicate $\mathsf{AppReturns}$ can thus be easily defined in terms of $\mathsf{AppEval}$, with a definition that need not refer to program values.

$$\mathsf{AppReturns}\, F\, V\, P \quad\equiv\quad \exists V'.\; \mathsf{AppEval}\, F\, V\, V' \;\wedge\; P\, V'$$

$$
\begin{array}{lcl}
[\![\, \hat{v} \,]\!]^{\Gamma} & \equiv & \mathbf{return}\ \lceil \hat{v} \rceil^{\Gamma} \\
[\![\, \hat{f}\ \hat{v}_1 \ldots \hat{v}_n \,]\!]^{\Gamma} & \equiv & \mathbf{app}\ \lceil \hat{f} \rceil^{\Gamma}\ \lceil \hat{v}_1 \rceil^{\Gamma} \ldots \lceil \hat{v}_n \rceil^{\Gamma} \\
[\![\, \mathsf{crash} \,]\!]^{\Gamma} & \equiv & \mathbf{crash} \\
[\![\, \mathsf{if}\ \hat{v}\ \mathsf{then}\ \hat{t}_1\ \mathsf{else}\ \hat{t}_2 \,]\!]^{\Gamma} & \equiv & \mathbf{if}\ \lceil \hat{v} \rceil^{\Gamma}\ \mathbf{then}\ [\![\hat{t}_1]\!]^{\Gamma}\ \mathbf{else}\ [\![\hat{t}_2]\!]^{\Gamma} \\
[\![\, \mathsf{let}\ x = \Lambda \overline{A}.\, \hat{t}_1\ \mathsf{in}\ \hat{t}_2 \,]\!]^{\Gamma} & \equiv & \mathbf{let}_{\overline{A}}\ X\ =\ [\![\hat{t}_1]\!]^{\Gamma}\ \mathbf{in}\ [\![\hat{t}_2]\!]^{(\Gamma, x \mapsto X)} \\
[\![\, \mathsf{let\,rec}\ f = \Lambda \overline{A}. \lambda x_1 \ldots x_n. \hat{t}_1\ \mathsf{in}\ \hat{t}_2 \,]\!]^{\Gamma} & \equiv & \mathbf{let\ rec}_{\overline{A}}\ F\ X_1 \ldots X_n = [\![\hat{t}_1]\!]^{(\Gamma, f \mapsto F, x_i \mapsto X_i)} \\
 & & \qquad \mathbf{in}\ [\![\hat{t}_2]\!]^{(\Gamma, f \mapsto F)}
\end{array}
$$

Figure 4.6: Characteristic formula generator (formal presentation)

4.4.3 Characteristic formula generation with notation

The characteristic formula generator can now be given a formal presentation in which types are made explicit and in which Caml values are reflected into Coq through calls to the decoding operator $\lceil \cdot \rceil$. In order to justify that characteristic formulae can be displayed like the source code, I proceed in two steps. First, I describe the characteristic formula generator in terms of an intermediate layer of notation (Figure 4.6). Then, I define the notation layer in terms of higher-order logic connectives and in terms of the predicate $\mathsf{AppReturns}_n$ (Figure 4.7). The contents of those figures simply refine the earlier informal presentation. From those definitions, it appears clearly that the size of a characteristic formula is linear in the size of the source code it describes.

Observe that the notation for function definitions relies on two auxiliary pieces of notation, introduced with the keywords **let fun** and **body**. Those auxiliary definitions will be helpful later on for pretty-printing top-level functions and mutually-recursive functions.

The CFML generator annotates characteristic formulae with tags in order to ease the work of the Coq pretty-printer. A tag is simply an identity function with a particular name. For example, `tag_if` is the tag used to label Coq expressions that correspond to the characteristic formula of a conditional expression. The Coq definition of the tag and the definition of the notation in which it is involved appear next.

```
Definition tag_if : forall A, A -> A := id.
Notation "'If' V 'Then' F1 'Else' F2" :=
  (tag_if (fun P => (V = true -> F1 P) /\ (V = false -> F2 P))).
```

The use of tags, which appear as head constant of characteristic formulae, significantly eases the task of the pretty-printer of Coq because each tag corresponds to exactly one piece of notation.

$$\begin{array}{l}
\mathbf{return}\ V \\
\quad \equiv\ \lambda P.\ P\,V \\
\mathbf{app}\ F\ V_1 \ldots V_n \\
\quad \equiv\ \lambda P.\ \mathsf{AppReturns}_n\,F\,V_1 \ldots V_n\,P \\
\mathbf{crash} \\
\quad \equiv\ \lambda P.\ \mathsf{False} \\
\mathbf{if}\ V\ \mathbf{then}\ \mathcal{F}\ \mathbf{else}\ \mathcal{F}' \\
\quad \equiv\ \lambda P.\ (V = \mathsf{true} \Rightarrow \mathcal{F}\,P) \wedge (V = \mathsf{false} \Rightarrow \mathcal{F}'\,P) \\
\mathbf{let}_{\overline{A}}\ X\ =\ \mathcal{F}\ \mathbf{in}\ \mathcal{F}' \\
\quad \equiv\ \lambda P.\ \exists P'.\ (\forall \overline{A}.\ \mathcal{F}\,(P'\,\overline{A}) \wedge (\forall X.\ (\forall \overline{A}.\ P'\,\overline{A}\,(X\,\overline{A})) \Rightarrow \mathcal{F}'\,P) \\
\mathbf{let\ rec}_{\overline{A}}\ F\ X_1 \ldots X_n = \mathcal{F}_1\ \mathbf{in}\ \mathcal{F}_2 \\
\quad \equiv\ (\mathbf{let\ fun}\ F\ =\ (\mathbf{body}_{\overline{A}}\ F\ X_1 \ldots X_n\ =\ \mathcal{F}_1)\ \mathbf{in}\ \mathcal{F}_2) \\
\mathbf{let\ fun}\ F\ =\ \mathcal{H}\ \mathbf{in}\ \mathcal{F} \\
\quad \equiv\ \lambda P.\ \forall F.\ \mathcal{H} \Rightarrow \mathcal{F}\,P \\
\mathbf{body}_{\overline{A}}\ F\ X_1 \ldots X_n\ =\ \mathcal{F} \\
\quad \equiv\ \forall \overline{A}\,K.\ \mathsf{is_spec}_n\,K \wedge (\forall X_1 \ldots X_n.\ K\,X_1 \ldots X_n\,\mathcal{F}) \Rightarrow \mathsf{Spec}_n\,F\,K
\end{array}$$

Figure 4.7: Syntactic sugar to display characteristic formulae

4.5 Generated axioms for top-level definitions

Reasoning on complete programs Consider a program "$\mathsf{let}\,x = \hat{t}$". The CFML generator could produce a definition corresponding to the characteristic formula of the typed term $\hat{t}$, and then the user could use this definition to specify the result value x of the program with respect to some post-condition P.

$$\begin{array}{l}
\mathsf{Definition\ X_cf\ :=\ } [\![t]\!]. \\
\mathsf{Lemma\ X_spec\ :\ X_cf}\ P.
\end{array}$$

Yet, this approach is not really practical because a Caml program is typically made of a sequence of top-level declarations, not just of a single declaration.

Instead, the CFML generator produces axioms to describe the result values of each top-level declaration, and produces an axiom to describe the characteristic formula associated with each definition. In what follows, I first present the axioms that are generated for a top-level value definition, and then present specialized axioms for the case of functions.

Generated axioms for values Consider a top-level non-polymorphic definition "$\mathsf{let}\,x\ =\ \hat{t}^T$" that does not describe a function. The CFML tool generates two axioms. The first axiom, named X, has type $[\![T]\!]$, which is the Coq type that reflects

the type T. This axiom represents the result of the evaluation of the declaration "$\mathsf{let}\, x = \hat{t}$".

$$\mathsf{Axiom\ X\ :\ } \llbracket T \rrbracket.$$

The second axiom, named X_cf, can be used to establish properties about the value X. Given a post-condition P of type $\llbracket T \rrbracket \to \mathsf{Prop}$, this axiom describes what needs to be proved in order to deduce that the proposition $P\,\mathsf{X}$ holds.

$$\mathsf{Axiom\ X\_cf\ :\ } \forall P.\, [\![\, \hat{t} \,]\!]^{\emptyset}\, P \Rightarrow P\,\mathsf{X}.$$

When the definition x has a polymorphic type S, the Coq value X admits the type $\llbracket S \rrbracket$, which is of the form $\forall \overline{A}.\llbracket T \rrbracket$. In this case, the post-condition P has type $\forall \overline{A}.(\llbracket T \rrbracket \to \mathsf{Prop})$ and the axiom X_cf is as follows.

$$\mathsf{Axiom\ X\_cf\ :\ } \forall \overline{A}.\, \forall P.\ [\![\, \hat{t} \,]\!]^{\emptyset}\, (P\,\overline{A}) \ \Rightarrow\ (P\,\overline{A})\,(\mathsf{X}\,\overline{A}).$$

Proof that types are inhabited A declaration of the form "Axiom X : $\llbracket S \rrbracket$" would be incoherent if the type $\llbracket S \rrbracket$ were not inhabited. To avoid this issue, CFML generates a proof obligation to ensure that $\llbracket S \rrbracket$ is inhabited before producing the axiom X. The statement relies on the Coq predicate Inhab, which holds only of inhabited types. The lemma generated thus takes the form:

$$\mathsf{Lemma\ X\_safe\ :\ Inhab}\, \llbracket S \rrbracket.$$

This lemma can be discharged automatically by Coq whenever the type $\llbracket S \rrbracket$ is inhabited, through invokation of Coq's proof search tactic eauto. However, if the type $\llbracket S \rrbracket$ is not inhabited, then the lemma X_safe cannot be proved. In this case, the generated file does not compile, and soundness is not compromised.

Overall, CFML rejects definitions of values whose type is not inhabited. Fortunately, those values are typically uninteresting to specify. Indeed, a term can admit an un-inhabited type only if it never returns a value, that is, if it always either crashes or diverges.

Axioms generated for functions Consider a top-level function definition, of the form "$\mathsf{let\,rec}\, f\ =\ \lambda x.\, t$". Such a definition can be encoded as the definition "$\mathsf{let}\, f\ =\ (\mathsf{let\,rec}\, f\ =\ \lambda x.\, t\,\mathsf{in}\, f)$", which can be handled like other top-level value definitions. That said, it is more convenient in practice to directly produce an axiom corresponding to the body description of the function "$\mathsf{let\,rec}\, f = \lambda x.\, t$".

Consider a function definition of the form "$\mathsf{let\,rec}\, f = \Lambda \overline{A}.\lambda x_1 \ldots x_n.\hat{t}$". An axiom named F represents the function. The type Func can be safely considered as being inhabited, so there is no concern about soundness here.

$$\mathsf{Axiom\ F\ :\ Func.}$$

The axiom F_cf generated for F asserts the body description of the function f.

$$\mathsf{Axiom\ F\_cf\ :\ } \mathbf{body}_{\overline{A}}\, F\, X_1 \ldots X_n\ =\ [\![\, \hat{t} \,]\!]^{(f \mapsto F, x_i \mapsto X_i)}.$$

In Chapter 6, I prove that the all the generated axioms are sound.

4.6 Extensions

In this section, I explain how to extend the generation of characteristic formulae to handle mutually-recursive functions, assertions, and pattern matching. For the sake of presentation, I return to an informal presentation style, working with untyped terms.

4.6.1 Mutually-recursive functions

To reason on mutually-recursive functions, one needs to state the specification of each of the functions involved, and then to prove those functions correct together, because the termination of a function may depend on the termination of the other functions. Consider the definition of two mutually-recursive functions f_1 and f_2, of body t_1 and t_2. The corresponding characteristic formula is built as follows.

$$\begin{array}{rcl} [\![\mathsf{let}\ f_1 = \lambda x_1.\, t_1\ \mathsf{and}\ f_2 = \lambda x_2.\, t_2\ \mathsf{in}\ t]\!] & \equiv & \lambda P.\ \forall f_1. \forall f_2.\ \mathcal{H}_1 \wedge \mathcal{H}_2 \Rightarrow [\![t]\!]\, P \\ \text{where } \mathcal{H}_1 & \equiv & \forall K.\ \mathsf{is\_spec}\, K \wedge (\forall x_1.\ K\, x_1\, [\![t_1]\!]) \Rightarrow \mathsf{Spec}\, f_1\, K \\ \mathcal{H}_2 & \equiv & \forall K.\ \mathsf{is\_spec}\, K \wedge (\forall x_2.\ K\, x_2\, [\![t_2]\!]) \Rightarrow \mathsf{Spec}\, f_2\, K \end{array}$$

There might be more than two mutually-recursive functions, moreover each function might expect several arguments. To avoid the need to define an exponential amount of notation, I rely directly on the notation **body** for pretty-printing the body description $\mathcal{H}_i$ associated with each function, and I rely directly on a generalized version of the notation **let fun** for pretty-printing the definition of mutually-recursive definitions. For example, for the arity two, I define:

$$(\mathbf{let\ fun}\ F_1 = \mathcal{H}_1\ \mathbf{and}\ F_2 = \mathcal{H}_2\ \mathbf{in}\ \mathcal{F}) \quad \equiv \quad \lambda P.\ \forall F_1. \forall F_2.\ \mathcal{H}_1 \wedge \mathcal{H}_2 \Rightarrow \mathcal{F}\, P$$

Overall, I need one version of **body** for each possible number of arguments and one version of **let fun** for each possible number of mutually-recursive functions.

For top-level mutually-recursive functions, I first generate one axiom to represent each function, and then generate the body description of each of the functions. Those body descriptions may refer to the name of any of the mutually-recursive functions involved.

This approach to treating mutually-recursive functions works very well in practice, as I could experience through the verification of Okasaki's "bootstrapped queues", whose implementation relies on five mutually-recursive functions. Those functions involve polymorphic recursion and some of them expect several arguments.

4.6.2 Assertions

Programmers can use assertions to test at runtime whether a particular property or invariant holds of the data structures being manipulated is satisfied. If an assertion evaluates to the boolean `true`, then the program execution continues as if the assertion was not present. However, if the assertion evaluates to `false`, then the program

halts by raising a fatal error. Assertions are thus a convenient tool for debugging, as they enable the programmer to detect precisely when and where the specifications are violated.

When formal methods are used to prove that the program satisfies its specification before the program is executed, the use of assertions appears to be of much less interest. Nevertheless, I want to be able to verify existing programs, whose code might already contain assertions. So, I wish to prove that all the assertions contained in a program always evaluate to true. In what follows, I recall the syntax and semantics of assertions and I explain how to build characteristic formulae for assertions.

The term "$\mathsf{assert}\ t_1\ \mathsf{in}\ t_2$" evaluates the assertion t_1 and proceeds to the evaluation of t_2 only when t_1 returns true.

$$\frac{t_1 \Downarrow \mathsf{true} \qquad t_2 \Downarrow v}{(\mathsf{assert}\ t_1\ \mathsf{in}\ t_2) \Downarrow v}$$

The characteristic formula is very simple to build. Indeed, to show that the term "$\mathsf{assert}\ t_1\ \mathsf{in}\ t_2$" admits P as post-condition, it suffices to show that the characteristic formula of t_1 holds of the post-condition "$= \mathsf{true}$" and that t_2 admits P as post-condition. Hence the following definition:

$$[\![\mathsf{assert}\ t_1\ \mathsf{in}\ t_2]\!] \quad \equiv \quad \lambda P.\ [\![t_1]\!]\,(= \mathsf{true}) \wedge [\![t_2]\!]\,P$$

Remark: the expression "$\mathsf{assert}\ t_1\ \mathsf{in}\ t_2$" has exactly the same behavior as the expression "$\mathsf{let}\, x = t_1\, \mathsf{in}\, \mathsf{if}\, x\, \mathsf{then}\, t_2\, \mathsf{else}\, \mathsf{crash}$". One can indeed prove that the characteristic formula of this latter term is equivalent to the formula given above.

4.6.3 Pattern matching

Syntax and semantics of pattern-matching The grammar of terms is extended with a construction "$\mathsf{match}\, v\, \mathsf{with}\, p_1 \mapsto t_1 \mid \ldots \mid p_n \mapsto t_n$", denoting that the value v is matched against the linear patterns p_i. The terms t_i are the continuations associated with each of the patterns. In the formal presentation, I let b range over pairs of a pattern and a continuation, of the form $p \mapsto t$. Formally, the pattern matching construction takes the form "$\mathsf{match}\, v\, \mathsf{with}\, \overline{b}$".

$$\begin{array}{lcl} t & := & \ldots \quad \mid \quad \mathsf{match}\, v\, \mathsf{with}\, \overline{b} \\ b & := & p \mapsto t \\ p & := & x \quad \mid \quad n \quad \mid \quad D(p, \ldots, p) \end{array}$$

For generating characteristic formulae, it is simpler that patterns do not contain any wildcard. So, during the normalization process of the source code, I replace all wildcards by fresh identifiers. Note: the treatment of and-patterns, or-patterns, and when-clauses is not described.

The big-step reduction rules associated with patterns are given next. Below, the function fv is used to compute the set of free variables of a pattern.

$$\frac{\overline{x} = \mathsf{fv}\, p \qquad v = [\overline{x} \to \overline{v}]\, p \qquad ([\overline{x} \to \overline{v}]\, t) \Downarrow v}{(\mathsf{match}\ v\ \mathsf{with}\ p \mapsto t \mid \overline{b}) \Downarrow v}$$

$$\frac{\overline{x} = \mathsf{fv}\, p \qquad (\forall \overline{v_i}.\ v \neq [\overline{x} \to \overline{v}]\, p) \qquad (\mathsf{match}\ v\ \mathsf{with}\ \overline{b}) \Downarrow v}{(\mathsf{match}\ v\ \mathsf{with}\ p \mapsto t \mid \overline{b}) \Downarrow v}$$

Example I first illustrate the generation of characteristic formulae for pattern-matching through an example. Assume that v is a pair of integers being pattern-matched first against the pattern $(3, x)$ and then against the pattern $(x, 4)$. Assume that the two associated continuation are terms named t_1 and t_2. The corresponding characteristic formula is described by the following statement.

$$\begin{aligned}[\![\mathsf{match}\ v\ \mathsf{with}\ (3,x) \mapsto t_1 \mid (x,4) \mapsto t_2]\!] \equiv\ \lambda P.\quad & (\forall x.\ v = (3,x) \Rightarrow [\![t_1]\!]\ P) \\ & \wedge ((\forall x.\ v \neq (3,x)) \Rightarrow \\ & \qquad (\forall x.\ v = (x,4) \Rightarrow [\![t_2]\!]\ P) \\ & \qquad \wedge ((\forall \overline{x}.\ v \neq (x,4) \Rightarrow \mathsf{False}))\end{aligned}$$

If the value v is equal to $(3, x)$ for some x, then we have to show that t_1 returns a value satisfying P. Otherwise, we can assume that the value v is different from $(3, x)$ for all x. Then, if the value v is equal to $(4, x)$ for some x, we have to show that t_2 returns a value satisfying P. Otherwise, we can assume that, the value v is different from $(4, x)$ for all x. Since there are no remaining cases in the pattern-matching, and since we want to show that the code does not crash, we have to show that all the possible cases have been already covered. Hence the proof obligation False.

Observe that the treatment of pattern matching in characteristic formulae allows for reasoning on the branches one by one, accumulating the negation of all the previous patterns which have not been matched.

In practice, I pretty-print the above characteristic formula as follows:

$$\begin{aligned}&\textbf{case}\ v = (3,x)\ \textbf{vars}\ x\ \textbf{then}\ [\![t_1]\!]\ \textbf{else} \\ &\textbf{case}\ v = (x,4)\ \textbf{vars}\ x\ \textbf{then}\ [\![t_2]\!]\ \textbf{else crash}\end{aligned}$$

The idea is to rely on a cascade of cases, introduced by the keyword **case**, corresponding to the various patterns being tested. The keyword **vars** introduces the list of pattern variables bound by each pattern. The **then** keyword introduces the characteristic formula associated with continuation to be executed when the current pattern matches. The **else** keyword introduces the formula associated with the continuation to be followed otherwise. Note that variables bound by the keyword **vars** range over the **then** branch but not over the **else** branch. Those piece of notation are formally defined soon afterwards.

Informal presentation Consider a term "$\mathsf{match}\ v\ \mathsf{with}\ p \mapsto t \mid \overline{b}$". The value v is being matched against a pattern p, associated with a continuation t, and let $\overline{b}$ range over the remaining branches of the pattern-matching on v. Let $\overline{x}$ denote the set of pattern variables occurring in p. The generation of characteristic formulae for pattern-matching is described through the following two rules.

$$\begin{array}{lcll} [\![\mathsf{match}\ v\ \mathsf{with}\ p \mapsto t \mid \overline{b}]\!] & \equiv & \lambda P. & \big(\forall \overline{x}.\ v = p \ \Rightarrow\ [\![t]\!]\ P\big) \\ & & \wedge & \big((\forall \overline{x}.\ v \neq p) \ \Rightarrow\ [\![\mathsf{match}\ v\ \mathsf{with}\ \overline{b}]\!]\ P\big) \\ [\![\mathsf{match}\ v\ \mathsf{with}\ \emptyset]\!] & \equiv & \lambda P.\ \mathsf{False} & \end{array}$$

Remark: The value v describing the argument of the pattern matching is duplicated in each branch. Thus, the size of the characteristic formula might not be linear in the size of the input program when the value v is not reduced to a variable. In practice, this does not appear to be a problem. However, from a theoretical perspective, it is straightforward to re-establish linearity of the characteristic formula. Indeed, it suffices to change the term "$\mathsf{match}\ v\ \mathsf{with}\ p \mapsto t \mid \overline{b}$" into "$\mathsf{let}\ x = v\ \mathsf{in}\ \mathsf{match}\ x\ \mathsf{with}\ p \mapsto t \mid \overline{b}$" for some fresh name x before computing the characteristic formula.

Decoding of patterns In order to formalize the generation of characteristic formulae for patterns, I need to consider typed patterns, written $\hat{p}$, and to define the decoding of a typed pattern. This function, written $\lceil \hat{p} \rceil^{\Gamma}$, transforms a typed pattern $\hat{p}$ of type T into a Coq value of type $[\![T]\!]$, by replacing all data constructors with their logical counterpart, and by replacing pattern variables with their associated Coq variable from the map Γ.

$$\begin{array}{lcl} \lceil x \rceil^{\Gamma} & \equiv & \Gamma(x) \\ \lceil n \rceil^{\Gamma} & \equiv & n \\ \lceil D\,\overline{T}(\hat{p}_1, \ldots, \hat{p}_n) \rceil^{\Gamma} & \equiv & D\,[\![\overline{T}]\!]\,(\lceil \hat{p}_1 \rceil^{\Gamma}, \ldots, \lceil \hat{p}_n \rceil^{\Gamma}) \end{array}$$

Remark: the variables bound in Γ always have a monomorphic type, since pattern variables are not generalized (I have not included the construction "$\mathsf{let}\ p = t\ \mathsf{in}\ t$" in the source language).

Formal presentation, with notation I define the generation of characteristic formulae through an intermediate layer of syntactic sugar, relying on the notation **case** introduced earlier on in the example. Below, $\overline{x}$ denotes the list of variables bound by the pattern $\hat{p}$, and $\overline{X}$ denotes a list of fresh Coq variables of the same length.

$$\begin{array}{lcl} [\![\mathsf{match}\ \hat{v}\ \mathsf{with}\ \hat{p} \mapsto \hat{t} \mid \overline{b}]\!]^{\Gamma} & \equiv & \mathbf{case}\ \lceil \hat{v} \rceil^{\Gamma} = \lceil \hat{p} \rceil^{(\overline{x} \mapsto \overline{X})}\ \mathbf{vars}\ \overline{X} \\ & & \quad \mathbf{then}\ [\![\hat{t}]\!]^{(\Gamma, \overline{x} \mapsto \overline{X})} \\ & & \quad \mathbf{else}\ [\![\mathsf{match}\ \hat{v}\ \mathsf{with}\ \overline{b}]\!]^{\Gamma} \\ [\![\mathsf{match}\ \hat{v}\ \mathsf{with}\ \emptyset]\!]^{\Gamma} & \equiv & \mathbf{crash} \end{array}$$

where the notation **case** is formally defined as follows:

$$\begin{array}{l}(\textbf{case}\ V = V'\ \textbf{vars}\ \overline{X}\ \textbf{then}\ \mathcal{F}\ \textbf{else}\ \mathcal{F}') \quad \equiv \\ \quad \lambda P.\ (\forall \overline{X}.\ V = V' \Rightarrow \mathcal{F}\,P)\ \wedge\ ((\forall \overline{X}.\ V \neq V') \Rightarrow \mathcal{F}'\,P)\end{array}$$

Remark: in the characteristic formula, the value $\lceil \hat{v} \rceil^{\Gamma}$ appears under the scope of the pattern variables $\overline{x}$, although this was not the case in the program source code. So, if the value $\hat{v}$ contains a variable with the same name as a variable bound in the pattern $\hat{p}$, then we need to alpha-rename the pattern variable in conflict. Alternatively, we may first bind the value $\hat{v}$ to a fresh variable x, building the term "$\mathsf{let}\, x = \hat{v}\, \mathsf{in}\, \mathsf{match}\, x\, \mathsf{with}\, \hat{p} \mapsto \hat{t} \mid \overline{b}$", in which the value being matched, namely x, is guaranteed to be distinct from all the pattern variables. The implementation of CFML performs this transformation whenever the value $\hat{v}$ contains a free variable that clashes with one of the pattern variables involved in the pattern matching.

Exhaustive patterns By default, the OCaml type-checker checks for exhaustiveness of coverage in pattern-matching constructions, through an analysis based on types. If there exists a value that might not match any of the patterns, then the Caml compiler issues a warning. There are cases where the invariants of the program guarantee that the given set of branches actually covers all the possible cases. In such a case, the user may ignore the warning produced by the compiler.

When the compiler's analysis recognizes a pattern matching as exhaustive, then it is safe to modify the characteristic formula so as to remove the proof-obligation that corresponds to exhaustiveness. In such a case, I replace the formula **crash**, which is defined as "$\lambda P.\, \mathsf{False}$", with the formula **done**, which is defined as "$\lambda P.\, \mathsf{True}$". Through this change, a nontrivial proof obligation "**crash** P" is replaced with a trivial proof obligation "**done** P". In practice, most patterns are exhaustive, so the amount of work required in interactive proofs is greatly reduced by this optimization which exploits the strength of the exhaustiveness decision procedure.

Conclusion and extensions In summary, characteristic formulae rely on equalities and disequalities to reason on pattern matching. The branches of a pattern can be studied one by one, following the execution flow of the program. The CFML library also includes a tactic that applies to the characteristic formula of a pattern with n branches and directly generates n subgoals, plus one additional subgoal when the pattern is not recognized as exhaustive. The treatment of for alias patterns (of the form "$p\,\mathsf{as}\,x$") and of top-level or-patterns is not described here but is implemented in CFML. The treatment of when-clauses and general or-patterns is left to future work.

4.7 Formal proofs with characteristic formulae

In this section, I explain how to reason about programs through characteristic formulae. While it is possible to manipulate characteristic formulae directly, I have

found it much more effective to design a set of high-level tactics. Moreover, the use of high-level tactics means that the user does not need to know about the details of the construction of characteristic formulae.

4.7.1 Reasoning tactics: example with let-bindings

Definition of the core tactic To start with, assume we want to show that a term "$\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2$" returns a value satisfying a predicate P. The goal to be established, "$[\![\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2]\!]\, P$", is equivalent to the proposition shown next.

$$\exists P'.\ [\![t_1]\!]\, P' \ \wedge\ (\forall X.\ P'\, X \ \Rightarrow\ [\![t_2]\!]^{(x \mapsto X)}\, P)$$

On this goal, a call to the tactic `xlet` with an argument P_1 instantiates P' as P_1 and produces two subgoals. The first subgoal, "$[\![t_1]\!]\, P_1$", corresponds to the body of the let-binding. The second subgoal consists in proving "$[\![t_2]\!]^{(x \mapsto X)}\, P$" in a context extended with a free variable named X and an hypothesis $P_1\, X$.

The Coq implementation of the tactic `xlet` is straightforward. First, it provides its argument `P1` as a witness, with a call to "`exists` $P1$". Then, it splits the conjunction, with the tactic `split`. Finally, in the second subgoal, it introduces X as well as the hypothesis about X, calling the tactic `intro` twice. The corresponding Coq tactic definition appears next.

```
Tactic Notation "xlet" constr(P1) :=
  exists P1; split; [ | intro; intro ].
```

Inference of specification The tactic `xlet` described so far requires the intermediate specification P_1 to be provided explicitly. Yet, it is very often the case that this post-condition can be automatically inferred when solving the goal "$[\![t_1]\!]\, P_1$". For example, if t_1 is the application of a function, then the post-condition P_1 gets unified with the post-condition of that function. So, I also define a tactic `xlet` that does not expect any argument, and that simply instantiates P_1 with a fresh unification variable. As soon as this unification variable is instantiated in the first subgoal $[\![t_1]\!]\, P_1$, the hypothesis "$P_1\, X$" from the second subgoal becomes fully determined.

The implementation of this alternative tactic `xlet` relies on a call to the tactic "`exists_`", where the underscore symbol indicates that a fresh unification variable should be created.

```
Tactic Notation "xlet" :=
   exists _; split; [ | intro; intro ].
```

Generation of names Choosing appropriate names for variables and hypotheses is extremely important in practice for conducting interactive proofs. Indeed, a proof script that relies on generated names such as `H16` is very brittle, because any change to the program or to the proof may shift the names and result in `H16` being renamed, for example into `H19`. Moreover, proof obligations are much easier to read when they

involve only meaningful names. I have invested particular care in developing two versions of every tactic: one version where the user explicitly provides a name for a variable or an hypothesis, and one version that tries and picks a clever name automatically. As a result, it is possible to obtain robust proof scripts and readable proof obligations at a reasonable cost, by providing only a few names explicitly.

Let me illustrate this with the tactic `xlet`. A call to the tactic `intro` moves a universally-quantified variable or a hypothesis from the head of the goal into the proof context. The name X that appears in the characteristic formula of a let-binding comes from the source code of the program, after it has been put in normal form. When the name X comes from the original source code, it makes sense to reuse that name. However, when X corresponds to a variable that has been introduced during the normalization process (for assigning names to intermediate expressions from the source code), it is sometimes better to use a more meaningful name than the one that has been generated automatically. A call of the form "`xlet as X`" enables one to specify the name that should be used for X.

The tactic `xlet` also needs to come up with a name for the hypothesis "$P_1\,X$". By default, the tactic picks the name `HX`, which is obtained by placing the letter `H` in front of the name of the variable. However, it is also possible to specify the name to be used explicitly, by providing it as extra argument to `xlet`. This possibility is particularly useful when the hypothesis "$P_1\,X$" needs to be decomposed in several conjuncts. For example, if the post-condition P_1 of the term t_1 takes the form "$(\lambda X.\ \exists Y.\ H_1\,Y \wedge H_2\,Y)$", then a call to the tactic "`xlet as X (Y&M1&M2)`" leaves in the second subgoal a proof context containing a variable `Y`, an hypothesis `M1` of type "$H_1\,Y$" and an hypothesis `M2` of type "$H_2\,Y$". Without such a convenient feature, one would have to write instead something of the form "`xlet as X. destruct HX as (Y&M1&M2)`", or to rely on a more evolved tactic language such as SSReflect [30].

Overall, there are five ways to call the tactic `xlet`, as summarized below.

```
xlet.
xlet P1 as X.
xlet P1 as X HX.
xlet as X.
xlet as X HX.
```

All of the tactics implemented in CFML offer a similar degree of flexibility.

4.7.2 Tactics for reasoning on function applications

The process of proving a specification for a function f and then exploiting that specification for reasoning on applications of that function is as follows. First, one states the specification of f as a lemma of the form $\mathsf{Spec}_n\,f\,K$. Second, one proves this lemma using the characteristic formula describing the behavior of f. Third, one registers that lemma in a database of specification lemmas, so that the name of the lemma does not need to be mentioned explicitly in the proof scripts. Then, when reasoning on a piece of code that involves an application of the function f, a

proof obligation of the form $\mathsf{AppReturns}_m\, f\, x_1\, \ldots\, x_m\, P$ is involved. The tactic `xapp` helps proving that goal. The implementation of the tactic depends on whether the function is applied to the exact number of arguments it expects ($m = n$), or whether a partial-application is involved ($m < n$), or whether an over-application is involved ($m > n$). Note that all the lemmas about Spec_n mentioned thereafter are proved in Coq.

Normal applications The elimination lemma for the predicate Spec_n (introduced in §4.2.3) can be used to deduce a proposition about $\mathsf{AppReturns}_n\, f$ from the specification $\mathsf{Spec}_n\, f\, K$. However, it does not directly have a conclusion of the form $\mathsf{AppReturns}_m\, f\, x_1\, \ldots\, x_n\, P$. So, the implementation of the tactic `xapp` relies on a corollary of the elimination lemma, shown next.

$$\begin{cases} \mathsf{Spec}_n\, f\, K \\ \forall R.\ \mathsf{Weakenable}\, R \Rightarrow K\, x_1 \ldots x_n\, R \Rightarrow R\, P \end{cases} \quad \Rightarrow\ \mathsf{AppReturns}_n\, f\, x_1 \ldots x_n\, P$$

Let me illustrate the working of this lemma through an example. Suppose that a value x is a non-negative multiple of 4 (i.e. $x = 4 * k$ for some $k > 0$), and let me show that the application of the function half to x returns an even value, that is, "$\mathsf{AppReturns}_1\, \mathsf{half}\, x\, \mathsf{even}$". The proof obligation is:

$$\forall R.\quad \mathsf{Weakenable}\, R \quad \Rightarrow \quad (\forall n \geq 0.\, x = 2 * n \Rightarrow R\, (= n)) \quad \Rightarrow \quad R\, \mathsf{even}$$

By instantiating the hypothesis with $n = 2 * k$ and checking that $x = 2 * n$, we derive the fact "$R\, (= 2 * k)$". The conclusion, namely "$R\, \mathsf{even}$", follows: since R is compatible with weakening, it suffices to check that the proposition "$\forall y.\ y = 2 * k \ \Rightarrow\ \mathsf{even}\, y$" holds.

Partial applications Partial application occurs when the function f is applied to fewer arguments than it normally expects. The result of such a partial application is another function, call it g, whose specification is an appropriate specialization of the specification of f. For example, if K is the specification of f, then the partial application of f to an argument x admits the specification $K\, x$. So, to show that the partial application of f to arguments $x_1 \ldots x_n$ satisfies a post-condition P, one needs to prove that the specification P is a consequence of the specification $K\, x_1 \ldots x_n$. The following result is exploited by the tactic `xapp` for reasoning on partial applications ($n > m$).

$$\begin{cases} \mathsf{Spec}_n\, f\, K \\ \forall g.\ \mathsf{Spec}_{n-m}\, g\, (K\, x_1 \ldots x_m) \Rightarrow P\, g \end{cases} \quad \Rightarrow \quad \mathsf{AppReturns}_m\, f\, x_1 \ldots x_m\, P$$

Over-applications Over-application occurs when the function f is applied to more arguments that it normally expects. This situation typically occurs with higher-order combinators, such as $\mathsf{compose}$, that return a function. The case of over-applications can be viewed as a particular case of a normal applications. Indeed, a

m-ary application can be viewed as a n-ary application that returns a function of "$m-n$" arguments. More precisely, when $m > n$, the goal $\mathsf{AppReturns}_m\, f\, x_1 \ldots x_m\, P$ can be rewritten into the goal shown below.

$$\mathsf{AppReturns}_n\, f\, x_1 \ldots x_n\, (\lambda g.\, \mathsf{AppReturns}_{m-n}\, g\, x_{n+1} \ldots x_m\, P)$$

Hence, the reasoning on over-applications relies on the following lemma.

$$\begin{array}{l} \left\{ \begin{array}{l} \mathsf{Spec}_n\, f\, K \\ \forall R.\ \mathsf{Weakenable}\, R \Rightarrow K\, x_1 \ldots x_n\, R \Rightarrow R\, (\lambda g.\, \mathsf{AppReturns}_{m-n}\, g\, x_{n+1} \ldots x_m\, P) \end{array} \right. \\ \Rightarrow\ \mathsf{AppReturns}_m\, f\, x_1 \ldots x_m\, P \end{array}$$

4.7.3 Tactics for reasoning on function definitions

Reasoning by weakening The tactic `xweaken` allows proving a specification $\mathsf{Spec}_n\, F\, K'$ from an existing specification $\mathsf{Spec}_n\, F\, K$. Using weakening, one may derive several specifications for a single function without verifying the code of that function more than once. The targeted specification K' must be weaker than the existing specification K in the sense that $K\, x_1 \ldots x_n\, R$ has to imply $K'\, x_1 \ldots x_n\, R$ for any predicate R compatible with weakening. Notice that K' has to be a valid specification in the sense that it should satisfy the predicate $\mathsf{is\_spec}_n$.

$$\begin{array}{l} \left\{ \begin{array}{l} \mathsf{Spec}_n\, f\, K \\ \forall x_1 \ldots x_n\, R.\ \mathsf{Weakenable}\, R \Rightarrow (K\, x_1 \ldots x_n\, R \Rightarrow K'\, x_1 \ldots x_n\, R) \\ \mathsf{is\_spec}_n\, K' \end{array} \right. \\ \Rightarrow\ \mathsf{Spec}_n\, f\, K' \end{array}$$

For example, from the specification of the function half we can deduce a weaker specification that captures the fact that when the input is of the form $4 * m$ then the output is a non-negative integer.

$$\mathsf{Spec}_1\, \mathsf{half}\, (\lambda x\, R.\ \forall m.\ m \geq 0 \Rightarrow x = 4 * m \Rightarrow R\, (\lambda y.\, y \geq 0))$$

In this case, the proof obligation is:

$$\begin{array}{l} \forall x\, R.\ \mathsf{Weakenable}\, R \Rightarrow \big(\forall n.\ n \geq 0 \Rightarrow x = 2 * n \Rightarrow R\, (= n)\big) \\ \Rightarrow \big(\forall m.\ m \geq 0 \Rightarrow x = 4 * m \Rightarrow R\, (\lambda y.\, y \geq 0)\big) \end{array}$$

Consider a particular value of x, R and m. The goal is to show $R\, (\lambda y.\, y \geq 0)$, and we have the assumptions $\mathsf{Weakenable}\, R$ and $\big(\forall n.\ n \geq 0 \Rightarrow x = 2 * n \Rightarrow R\, (= n)\big)$ and $m \geq 0$ and $x = 4 * m$. If we instantiate n with $2 * m$ in the second assumption about R, we get $R\, (= 2 * m)$. Since R is compatible with weakening, we can prove the goal $R\, (\lambda y.\, y \geq 0)$ simply by showing that any value y equal to $2 * m$ is non-negative. This property holds because m is non-negative.

Reasoning by induction The purpose of the tactic `xinduction` is to establish by induction that a function f admits a specification K. More precisely, it states that, to prove $\mathsf{Spec}_n\, f\, K$, it suffices to show that K is a correct specification for the application of f to an argument x under the assumption that K is already a correct specification for applications of f to smaller arguments.

In the particular case of a function of arity one, calling the tactic `xinduction` on an argument $(\prec)$, which denotes a well-founded relation used to argue for termination, transforms a goal of the form $\mathsf{Spec}_1\, f\, K$ into the goal:

$$\mathsf{Spec}_1\, f\, (\lambda x\, R.\ \mathsf{Spec}_1\, f\, (\lambda x'\, R'.\, x' \prec x \ \Rightarrow\ K\, x'\, R') \ \Rightarrow\ K\, x\, R)$$

The tactic `xinduction` relies on the following lemma.

$$\left(\begin{array}{l} \mathsf{Spec}_n\, f\, (\lambda x_1 \ldots x_n\, R. \\ \quad \mathsf{Spec}_n\, f\, (\lambda x'_1 \ldots x'_n\, R'.\, (x'_1, \ldots, x'_n) \prec (x_1, \ldots, x_n) \ \Rightarrow\ K\, x'_1 \ldots x'_n\, R') \\ \quad \Rightarrow\ K\, x_1 \ldots x_n\, R) \end{array}\right) \\ \Rightarrow\ \mathsf{Spec}_n\, f\, K$$

where f has type Func, K has type $A_1 \to \ldots A_n \to ((B \to \mathsf{Prop}) \to \mathsf{Prop}) \to \mathsf{Prop}$, and $(\prec)$ is a well-founded binary relation over values of type $A_1 \times \ldots \times A_n$.

Advanced reasoning by induction The induction lemma can be used to establish the specification of a function by induction on a well-founded relation over the arguments of that function. However, it does not support induction over the structure of the proof of an inductive proposition. The purpose of the tactic `xintros` is to change a goal of the form $\mathsf{Spec}_n\, f\, K$ into a goal with a conclusion about $\mathsf{AppReturns}_n\, f$, thereby making it possible to conduct advanced forms of induction.

The implementation of the tactic `xintros` relies on an introduction lemma for specifications, which is a form of reciprocal to the elimination lemma: it allows proving the specification of a function f in terms of a description of the behavior of applications of f to some arguments. This introduction lemma is formalized as follows.

$$\left\{\begin{array}{l} \forall x_1 \ldots x_n.\ K\, x_1 \ldots\, x_n\, (\mathsf{AppReturns}_n\, f\, x_1 \ldots x_n) \\ \mathsf{Spec}_n\, f\, (\lambda x_1 \ldots x_n\, R.\, \mathsf{True}) \\ \mathsf{is\_spec}_n\, K \end{array}\right. \quad \Rightarrow \quad \mathsf{Spec}_n\, f\, K$$

The lemma involves two side-conditions. The first one asserts that f is a n-ary curried function, meaning that the application to the $n-1$ first arguments always terminates. The second hypothesis asserts that K is a valid specification, in the sense that it is compatible with weakening.

4.7.4 Overview of all tactics

The CFML library includes one tactic for each construction of the language. I have already presented the tactic `xlet` for let-bindings and the tactic `xapp` for applications. Four other tactics are briefly described next. The tactic `xret` is used to reason

on a value. It simply unfolds the notation **return**, changing a goal "(**return** V) P" into "$P\,V$". The tactic `xif` helps reasoning on a conditional. It applies to a goal of the form "(**if** V **then** $\mathcal{F}$ **else** $\mathcal{F}'$) P" and produces two subgoals: first "$\mathcal{F}\ P$", with an hypothesis $V = \mathsf{true}$, and second "$\mathcal{F}\ P$", with an hypothesis $V = \mathsf{false}$. The syntax "`xif as H`" allows specifying how to name those new hypotheses. The tactic `xfun S` applies to a goal of the form "(**let fun** $F = \mathcal{H}$ **in** $\mathcal{F}$) P". It leaves two subgoals. The first goal requires proving the proposition S, which is typically of of the form $\mathsf{Spec}FK$, under the assumption $\mathcal{H}$, which is the body description for F. The second goal corresponds to the continuation: it requires proving $\mathcal{F}\,P$ under the assumption S, which can be used to reason about applications of the function F.

For pattern matching, I provide the tactic `xmatch` that produces one goal for each branch of the pattern matching. In case the completeness of the pattern matching could not be established automatically, the tactic produces an additional goal asserting that the pattern matching is complete (recall §4.6.3). I also provide a tactic, called `xcase`, for reasoning on the branches one by one. This strategy allows exploiting the assumption that a pattern has not been matched for deriving a fact that needs to be exploited through the reasoning on the remaining branches. Thereby, the tactic `xcase` helps factorizing pieces of proof involved in the verification of a nontrivial pattern matching.

The remaining tactics have already been described. They are briefly summarized next. The tactic `xweaken` allows establishing a specification from another specification, `xinduction` is used for proving a specification by induction on a well-founded relation, and `xintros` is used for proving a specification by induction on the proof of an inductively-defined predicate. The tactic `xcf` applies the characteristic formula of a top-level definition. The tactic `xgo` automatically applies the appropriate `x`-tactic, and stops when a specification of a local function is needed or when a ghost variable involved in an application cannot be inferred.

Chapter 5

Generalization to imperative programs

In this chapter, I explain how to generalize the ingredients from the previous chapter in order to support imperative programs. First, I describe a data structure for representing heaps, and define Separation Logic connectives for concisely describing those heaps. Second, I define a predicate transformer called local that supports in particular applications of the frame rule. Third, I introduce the predicates used to specify imperative functions. I then explain how to build characteristic formulae for imperative programs using those ingredients. Finally, I describe the treatment of null pointers and strong updates.

5.1 Extension of the source language

5.1.1 Extension of the syntax and semantics 对源语言语法和语义的拓展

To start with, I briefly describe the syntax and the semantics of the source language extended with imperative features. Compared with the pure language from the previous chapter, the grammar of values is extended with memory locations, written l. The null pointer is just a particular location that never gets allocated. Values are also extended with primitive functions for allocating, reading and updating reference cells, and with a primitive function for comparing pointers. The unit value, written tt, is defined as a particular algebraic data type definition with a unique data constructor. The grammar of terms is extended with sequence, for-loops and while-loops. The grammar of ML types is extended with reference types.

$$
\begin{array}{lcl}
l & := & \textit{locations} \\
v & := & \ldots \mid l \mid \mathsf{ref} \mid \mathsf{get} \mid \mathsf{set} \mid \mathsf{cmp} \\
t & := & \ldots \mid t\,;\,t \mid \mathsf{for}\,x = v\,\mathsf{to}\,v\,\mathsf{do}\,t \mid \mathsf{while}\,t\,\mathsf{do}\,t \\
\tau & := & \ldots \mid \mathsf{ref}\,\tau
\end{array}
$$

$$\frac{}{v_{/m} \Downarrow v_{/m}} \qquad \frac{([f \to \mu f.\lambda x.t]\,[x \to v]\,t)_{/m} \Downarrow v'_{/m'}}{((\mu f.\lambda x.t)\,v)_{/m} \Downarrow v'_{/m'}}$$

$$\frac{t_{1/m_1} \Downarrow tt_{/m_2} \qquad t_{2/m_2} \Downarrow v'_{/m_3}}{(t_1\,;\,t_2)_{/m_1} \Downarrow v'_{/m_3}} \qquad \frac{t_{1/m_1} \Downarrow v_{/m_2} \qquad ([x \to v]\,t_2)_{/m_2} \Downarrow v'_{/m_3}}{(\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2)_{/m_1} \Downarrow v'_{/m_3}}$$

$$\frac{t_{1/m} \Downarrow v'_{/m'}}{(\mathsf{if}\,\mathsf{true}\,\mathsf{then}\,t_1\,\mathsf{else}\,t_2)_{/m} \Downarrow v'_{/m'}} \qquad \frac{t_{2/m} \Downarrow v'_{/m'}}{(\mathsf{if}\,\mathsf{false}\,\mathsf{then}\,t_1\,\mathsf{else}\,t_2)_{/m} \Downarrow v'_{/m'}}$$

$$\frac{l = 1 + \mathsf{max}(\mathrm{dom}(m))}{(\mathsf{ref}\,v)_{/m} \Downarrow l_{/m \uplus [l \mapsto v]}} \qquad \frac{l \in \mathrm{dom}(m)}{(\mathsf{get}\,l)_{/m} \Downarrow (m[l])_{/m}} \qquad \frac{l \in \mathrm{dom}(m)}{(\mathsf{set}\,l\,v)_{/m} \Downarrow tt_{/(m[l \mapsto v])}}$$

$$\frac{l_1 = l_2}{(\mathsf{cmp}\,l_1\,l_2)_{/m} \Downarrow \mathsf{true}_{/m}} \qquad \frac{l_1 \neq l_2}{(\mathsf{cmp}\,l_1\,l_2)_{/m} \Downarrow \mathsf{false}_{/m}}$$

$$\frac{a \leq b \qquad ([i \to a]\,t)_{/m_1} \Downarrow tt_{/m_2} \qquad (\mathsf{for}\,i = a + 1\,\mathsf{to}\,b\,\mathsf{do}\,t)_{/m_2} \Downarrow tt_{/m_3}}{(\mathsf{for}\,i = a\,\mathsf{to}\,b\,\mathsf{do}\,t)_{/m_1} \Downarrow tt_{/m_3}}$$

$$\frac{a > b}{(\mathsf{for}\,i = a\,\mathsf{to}\,b\,\mathsf{do}\,t)_{/m} \Downarrow tt_{/m}} \qquad \frac{t_{1/m} \Downarrow \mathsf{false}_{/m'}}{(\mathsf{while}\,t_1\,\mathsf{do}\,t_2)_{/m} \Downarrow tt_{/m'}}$$

$$\frac{t_{1/m_1} \Downarrow \mathsf{true}_{/m_2} \qquad t_{2/m_2} \Downarrow tt_{/m_3} \qquad (\mathsf{while}\,t_1\,\mathsf{do}\,t_2)_{/m_3} \Downarrow tt_{/m_4}}{(\mathsf{while}\,t_1\,\mathsf{do}\,t_2)_{/m_1} \Downarrow tt_{/m_4}}$$

Figure 5.1: Semantics of imperative programs

Remark: I write the application of the primitive function set in the form "$\mathsf{set}\,l\,v$", as if set was a function of two arguments. However, it is simpler for the proof to view set as a function of one argument that expects a pair of values, as this avoids the need to consider partially-applied primitive functions.

A memory store, written m, maps locations to program values. The reduction judgment takes the form $t_{/m} \Downarrow v'_{/m'}$, asserting that the evaluation of the term t in a store m terminates and returns a value v' in a store m'. The definition of this big-step judgment is standard. In the rules shown in Figure 5.1, $m[l]$ denotes the value stored in m at location l, $m[l \mapsto v]$ describes a memory update at location l and $m \uplus [l \mapsto v]$ describes a memory allocation at location l. More generally, $m_1 \uplus m_2$ denotes the disjoint union of two stores m_1 and m_2.

5.1.2 Extension of weak-ML

In this section, I extend the definitions of the type system weak-ML to add support for imperative features. One of the key feature of weak-ML is that all locations admit a constant type, called loc, instead of a type of the form $\mathsf{ref}\,\tau$ that carries the type associated with the corresponding memory cell. The grammar of weak-ML types is thus extended as follows.

$$T \quad := \quad \ldots \quad | \quad \mathsf{loc}$$

The erasure operation from ML types into weak-ML types maps all types of the form $\mathsf{ref}\,\tau$ to the constant type loc.

$$\langle \mathsf{ref}\,\tau \rangle \quad \equiv \quad \mathsf{loc}$$

The notion of typed term and of typed value immediately extend to the imperative setting. Due to the value restriction, only variables immediately bound to a value may receive a polymorphic type. This restriction means that the grammar of typed terms includes bindings of the form "$\mathsf{let}\,x = \Lambda\overline{A}.\,\hat{v}_1\,\mathsf{in}\,\hat{t}_2$" as well as bindings of the form "$\mathsf{let}\,x = \hat{t}_1\,\mathsf{in}\,\hat{t}_2$", but not the general form "$\mathsf{let}\,x = \Lambda\overline{A}.\,\hat{t}_1\,\mathsf{in}\,\hat{t}_2$".

The typing judgment for imperative weak-ML programs takes the form $\Delta \vdash \hat{t}$. Contrary to the traditional ML typing judgment, the weak-ML typing judgment does not involve a store typing. Such an oracle is not needed because all locations admit the constant type loc. The typing judgment for imperative weak-ML programs thus directly extend the typing judgment for pure programs that was presented earlier on (§4.3.3), with the exception of the typing rule for let-bindings which is replaced by two rules so as to take into account the value restriction. The new rules appear next.

$$\frac{}{\Delta \vdash l^{\mathsf{loc}}} \qquad \frac{\Delta \vdash \hat{t}_1^{\mathsf{unit}} \qquad \Delta \vdash \hat{t}_2^{T}}{\Delta \vdash (\hat{t}_1^{\mathsf{unit}}\,;\,\hat{t}_2^{T})^{T}} \qquad \frac{\Delta \vdash \hat{t}_1^{T_1} \qquad \Delta, x : T_1 \vdash \hat{t}_2^{T_2}}{\Delta \vdash (\mathsf{let}\,x = \hat{t}_1^{T_1}\,\mathsf{in}\,\hat{t}_2^{T_2})^{T_2}}$$

$$\frac{\Delta, \overline{A} \vdash \hat{v}_1^{T_1} \qquad \Delta, x : \forall\overline{A}.T_1 \vdash \hat{t}_2^{T_2}}{\Delta \vdash (\mathsf{let}\,x = \Lambda\overline{A}.\,\hat{v}_1^{T_1}\,\mathsf{in}\,\hat{t}_2^{T_2})^{T_2}} \qquad \frac{\Delta \vdash \hat{t}_1^{\mathsf{bool}} \qquad \Delta \vdash \hat{t}_2^{\mathsf{unit}}}{\Delta \vdash (\mathsf{while}\,\hat{t}_1^{\mathsf{bool}}\,\mathsf{do}\,\hat{t}_2^{\mathsf{unit}})^{\mathsf{unit}}}$$

$$\frac{\Delta \vdash \hat{v}_1^{\mathsf{int}} \qquad \Delta \vdash \hat{v}_2^{\mathsf{int}} \qquad \Delta, i : \mathsf{int} \vdash \hat{t}_3^{\mathsf{unit}}}{\Delta \vdash (\mathsf{for}\,i = \hat{v}_1^{\mathsf{int}}\,\mathsf{to}\,\hat{v}_2^{\mathsf{int}}\,\mathsf{do}\,\hat{t}_3^{\mathsf{unit}})^{\mathsf{unit}}}$$

Two additional typing definitions are explained next. First, all primitive functions admit the type func, just like all other functions. Second, the value null, which is a particular location, admits the type loc. Observe that, since applications are unconstrained in weak-ML, applications of primitive functions for manipulating references are also unconstrained. This is how weak-ML accommodates strong updates.

In Coq, I introduce an abstract type called Loc for describing locations. The weak-ML type loc is reflected as the Coq type Loc. Note that a source program

$$\begin{array}{lcl} \varnothing & \equiv & \emptyset \\ l \rightarrow_{\mathcal{T}} V & \equiv & [l := (\mathcal{T}, V)] \\ h_1 + h_2 & \equiv & h_1 \cup h_2 \\ h_1 \perp h_2 & \equiv & (\mathrm{dom}(h_1) \cap \mathrm{dom}(h_2) = \emptyset) \end{array}$$

Figure 5.2: Construction of heaps in terms of operations on finite maps

does not contain any location, so decoding locations is not needed for computing the characteristic formula of a program.

$$[\![\mathsf{loc}]\!] \equiv \mathsf{Loc}$$

5.2 Specification of locations and heaps

5.2.1 Representation of heaps

I now explain how memory stores are described in the logic as values of type Heap. Whereas a memory store m maps locations to Caml values, a heap h maps locations to Coq values. Moreover, whereas a store describes the entire memory state, a heap may describe only a fragment of a memory state. Intuitively, a heap h is a Coq value that describes a piece of a memory store m if, for every location l from the domain of h, the value stored in h at location l is the Coq value that corresponds to the Caml value stored in m at location l. The connection between heaps and memory states is formalized later on, in Chapter 7. At this point, I only discuss the representation and specification of heaps.

The data type Loc represents locations. It is isomorphic to natural numbers. The data type Heap represent heaps. Heaps are represented as finite maps from locations to values of type Dyn, where Dyn is the type of pairs whose first component is a type $\mathcal{T}$ and whose second component is a value V of type $\mathcal{T}$.

$$\begin{array}{lcl} \mathsf{Heap} & \equiv & \mathsf{Fmap\ Loc\ Dyn} \\ \mathsf{Dyn} & \equiv & \Sigma_{\mathcal{T}}\, \mathcal{T} \end{array}$$

Note: in Coq, finite map can be represented using logical functions. More precisely, the datatype Heap is defined as the set of functions of type $\mathsf{Loc} \rightarrow \mathsf{option\ Dyn}$ that return a value different from None only for a finite number of arguments. Technically, $\mathsf{Heap} \equiv \{f : (\mathsf{Loc} \rightarrow \mathsf{option\ Dyn}) \mid \exists(L : \mathsf{list\ loc}).\ \forall(x : \mathsf{loc}).\ f\,x \neq \mathsf{None} \Rightarrow x \in L\}$.

Operations on heaps are defined in Figure 5.2 and explained next. 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 the location l to the Coq value V of type $\mathcal{T}$. 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, so it does not matter how the union operator is defined for maps with overlapping

$$
\begin{array}{lcll}
[\,] & \equiv & \lambda h.\ h = \varnothing & \\
[\mathcal{P}] & \equiv & \lambda h.\ h = \varnothing \wedge \mathcal{P} & \text{(where } \mathcal{P} \text{ is any proposition)} \\
l \hookrightarrow_{\mathcal{T}} V & \equiv & \lambda h.\ h = (l \rightarrow_{\mathcal{T}} V) & \\
H_1 * H_2 & \equiv & \lambda h.\ \exists h_1\, h_2.\ (h_1 \perp h_2) \ \wedge\ h = h_1 + h_2 \ \wedge\ H_1\, h_1 \ \wedge\ H_2\, h_2 & \\
\exists x.\, H & \equiv & \lambda h.\ \exists x.\, H\, h & \text{(where } x \text{ is bound in } H)
\end{array}
$$

Figure 5.3: Combinators for heap descriptions

domains. Finally, two heaps are said to be disjoint, written $h_1 \perp h_2$, when their underlying maps have disjoint domains. Note that the definition of heaps and of all predicates on heaps is entirely formalized in Coq.

5.2.2 Predicates on heaps

I now describe predicates for specifying heaps in Separation Logic style, closely following the definitions used in Ynot [17]. Heap predicates are simply predicates over values of type Heap. I use the letter H to range over heap predicates. For convenience, the type of such predicates is abbreviated as Hprop.

$$\mathsf{Hprop} \quad \equiv \quad \mathsf{Heap} \rightarrow \mathsf{Prop}$$

One major contribution of Separation Logic [77] is the separating conjunction (also called spatial conjunction). $H_1 * H_2$ describes a heap made of two disjoint parts such that the first one satisfies H_1 and the second one satisfies H_2. Compared with expressing properties on heaps directly in terms of heap representations, $H_1 * H_2$ concisely captures the disjointedness of the two heaps involved. Apart from the setting up of the core definition and lemmas of the CFML library, I always work in terms of heap predicates and never refer to heap representations directly.

Heap combinators, which are defined in Coq, appear in Figure 5.3. Empty heaps are characterized by the predicate $[\,]$. A singleton heap binding a location l to a value V of type $\mathcal{T}$ is characterized by the predicate $l \hookrightarrow_{\mathcal{T}} V$. Since the type $\mathcal{T}$ can be deduced from the value V, I often drop the type and write $l \hookrightarrow V$. The predicate $H_1 * H_2$ holds of a disjoint union of a heap satisfying H_1 and of a heap satisfying 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 only when the proposition $\mathcal{P}$ is 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.[1]

[1] The formal definition for existentials properly handles binders. It actually takes the form $\mathsf{hexists}\, J$, where J is a predicate on the value x. Formally:

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

I recall next the syntactic sugar introduced for post-conditions (§3.1). The post-condition $\#H$ describes a term that returns the unit value tt and produces a heap satisfying H. So, $\#H$ is a shortcut for $\lambda\_ : \mathsf{unit}.\, H$. The spatial conjunction of a post-condition Q with a heap satisfying H is written $Q \star H$, and is defined as $\lambda x.\, Q\, x * H$.

Finally, I rely on an entailment relation, written $H_1 \rhd H_2$, to capture that any heap satisfying H_1 also satisfies H_2.

$$H_1 \rhd H_2 \quad \equiv \quad \forall h.\ H_1\, h \Rightarrow H_2\, h$$

I also define a corresponding entailment relation for post-conditions. The proposition $Q_1 \blacktriangleright Q_2$ asserts that for any output value x and any output heap h, if $Q_1\, x\, h$ holds then $Q_2\, x\, h$ also holds. Entailment between post-conditions can be formally defined in terms of entailment on heap descriptions, as follows.

$$Q_1 \blacktriangleright Q_2 \quad \equiv \quad \forall x.\ Q_1\, x \rhd Q_2\, x$$

5.3 Local reasoning

In the introduction, I have suggested how to define a predicate called "frame" that applies to a characteristic formula and allows for applications of the frame rule while reasoning on that formula. In this section, I explain how to generalize the predicate "frame" into a predicate called "local" that also supports the rule of consequence as well as garbage collection. I then present elimination rules establishing that "local $\mathcal{F}$" is a formula where the frame rule, the rule of consequence and the rule of garbage collection can be applied an arbitrary number of times, in any order, before reasoning on the formula $\mathcal{F}$ itself.

5.3.1 Rules to be supported by the local predicate

The predicate "local" aims at simulating possible applications of the three following reasoning rules, which are presented using Hoare triples. In the frame rule, $H * H'$ extends the pre-condition H with a heap satisfying H' and $Q \star H'$ symmetrically extends the post-condition Q with a heap satisfying H'. The rule for garbage collection allows discarding a piece of heap H' from the pre-condition, as well as discarding a piece of heap H'' from the post-condition. The rule of consequence involves the entailment relation on heap predicates ($\rhd$) for strengthening the pre-condition and involve the entailment relation on post-conditions ($\blacktriangleright$) for strengthening the post-condition.

$$\frac{\{H\}\ t\ \{Q\}}{\{H * H'\}\ t\ \{Q \star H'\}}\ \textsc{frame} \qquad \frac{\{H\}\ t\ \{Q \star H''\}}{\{H * H'\}\ t\ \{Q\}}\ \textsc{gc}$$

$$\frac{H \rhd H' \qquad \{H'\}\ t\ \{Q'\} \qquad Q' \blacktriangleright Q}{\{H\}\ t\ \{Q\}}\ \textsc{consequence}$$

The first step towards the construction of the predicate local consists in combining the three rules into one. This is achieved through the rule shown next. In this rule, H and Q describe the outer pre- and post-condition, H_i and Q_f describe the inner pre- and post-condition, H_k correspond to the piece of heap being framed out, and H_g correspond to the piece of heap being discarded. Here and throughout the rest of the thesis, **i** stands for *initial*, **f** for *final*, **k** for *kept aside*, and **g** for *garbage*.

$$\frac{H \rhd H_i * H_k \qquad \{H_i\}\ t\ \{Q_f\} \qquad Q_f \star H_k \blacktriangleright Q \star H_g}{\{H\}\ t\ \{Q\}}\ \text{COMBINED}$$

One can check that this combined rule simulates the three previous rules.[2]

5.3.2 Definition of the local predicate

The predicate local corresponds to the predicate-transformer presentation of the combined reasoning rule, in the sense that the proposition "$\mathsf{local}\,\mathcal{F}\,H\,Q$" holds if one can find instantiations of H_i, H_k, H_g and Q_f such that the premises of the combined rule are satisfied, with the premise "$\{H_i\}\ t\ \{Q_f\}$" being replaced with "$\mathsf{local}\,\mathcal{F}\,H_i\,Q_f$". A first unsuccessful attempt at defining the predicate local is shown next, under the name local'.

$$\mathsf{local}'\,\mathcal{F} \quad \equiv \quad \lambda H\,Q.\ \exists H_i\,H_k\,H_g\,Q_f.\ \begin{cases} H \rhd H_i * H_k \\ \mathcal{F}\,H_i\,Q_f \\ Q_f \star H_k \blacktriangleright Q \star H_g \end{cases}$$

I explain soon afterwards why this definition is not expressive enough because it does not allow extracting existentials and propositions out of pre-conditions. For the time being, let me focus on explaining how to patch the definition of local' to obtain the correct definition of local. The idea is that we need to quantify over the heap h that satisfies the pre-condition H before quantifying existentially over the variables H_i, H_k, H_g and Q_f. So, the proposition "$\mathsf{local}\,\mathcal{F}\,H\,Q$" holds if, for any input heap h that satisfies the pre-condition H, the three following properties hold:

1. There exists a decomposition of the heap h as the disjoint union of a heap satisfying a predicate H_i and of a heap satisfying a predicate H_k.
2. The formula $\mathcal{F}$ holds of the pre-condition H_i and of some post-condition Q_f.
3. The post-condition $Q_f \star H_k$ entails the post-condition $Q \star H_g$ for some predicate H_g.

The formal definition of the predicate local appears next. It applies to a formula $\mathcal{F}$ with a type of the form $\mathsf{Hprop} \to (B \to \mathsf{Hprop}) \to \mathsf{Prop}$ for some type B. Recall

[2] For the frame rule, instantiate H_g as the empty heap predicate. For the rule of consequence, instantiate both H_k and H_g as the empty heap predicate. Finally, for the rule of garbage collection, instantiate H_i as H, H_k as H', Q_f as $Q * H''$, and H_g as $H' * H''$.

that H_i is the "initial" heap, Q_f describes the "final" post-condition, H_k is the heap being "kept aside", and H_g is the "garbage" heap.

$$\mathsf{local}\,\mathcal{F} \quad \equiv \quad \lambda H\,Q.\ \forall h.\ H\,h \Rightarrow \exists H_i\,H_k\,H_g\,Q_f.\ \begin{cases} (H_i * H_k)\,h \\ \mathcal{F}\,H_i\,Q_f \\ Q_f \star H_k \blacktriangleright Q \star H_g \end{cases}$$

Notice that the definition of local refers to a heap representation h. This heap representation never needs to be manipulated directly in proofs, as all the work can be conducted through the high-level elimination lemmas that are explained in the rest of this section.

Remark: the definition of the predicate local shows some similarities with the definition of the "STsep" monad from Hoare Type Theory [61], in the sense that both aim at baking the Separation Logic frame condition into a system defined in terms of heaps describing the whole memory.

5.3.3 Properties of local formulae

The first useful property about the predicate transformer local is that it may be ignored during reasoning. More precisely, if the goal is to prove "$\mathsf{local}\,\mathcal{F}\,H\,Q$", then it suffices to prove "$\mathcal{F}\,H\,Q$". (To check this implication, instantiate H_i as H, Q_f as Q, and H_k and H_g as the empty heap predicate.) Formally:

$$\forall H\,Q.\quad \mathcal{F}\,H\,Q \quad \Rightarrow \quad \mathsf{local}\,\mathcal{F}\,H\,Q$$

Another key property of local is its idempotence: iterated applications of local are redundant. In other words, a proposition of the form "$\mathsf{local}\,(\mathsf{local}\,\mathcal{F})\,H\,Q$" is always equivalent to the proposition "$\mathsf{local}\,\mathcal{F}\,H\,Q$". With predicate extensionality, the idempotence property can be stated very concisely, as follows.

$$\forall \mathcal{F}.\quad \mathsf{local}\,\mathcal{F} = \mathsf{local}\,(\mathsf{local}\,\mathcal{F})$$

It remains to explain how to exploit the predicate local in reasoning on characteristic formulae. Let me start with the case of the frame rule. The following statement is a direct consequence of the definition of the predicate local.

$$\mathcal{F}\,H_1\,Q_1 \quad \Rightarrow \quad \mathsf{local}\,\mathcal{F}\,(H_1 * H_2)\,(Q_1 \star H_2)$$

Yet, applying this lemma would result in consuming the occurrence of local at the head of the formula, preventing us from subsequently applying other rules. Fortunately, thanks to the idempotence of the predicate local, it is possible to derive a lemma where the local modifier is preserved.

$$\mathsf{local}\,\mathcal{F}\,H_1\,Q_1 \quad \Rightarrow \quad \mathsf{local}\,\mathcal{F}\,(H_1 * H_2)\,(Q_1 \star H_2)$$

In practice, I have found it more convenient with respect to tactics to reformulate the lemma in the following form.

$$\mathsf{is\_local}\,\mathcal{F} \quad \wedge \quad \mathcal{F}\,H_1\,Q_1 \quad \Rightarrow \quad \mathcal{F}\,(H_1 * H_2)\,(Q_1 \star H_2)$$

where the proposition "$\mathsf{is\_local}\,\mathcal{F}$" captures the fact that the formula $\mathcal{F}$ is equivalent to "$\mathsf{local}\,\mathcal{F}$". Formally,

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

A formula that satisfies the predicate $\mathsf{is\_local}$ is called a *local formula*. Observe that, due to the idempotence of local, any proposition of the form "$\mathsf{local}\,\mathcal{F}$" is a local formula. This result is formally stated as follows.

$$\forall \mathcal{F}.\ \mathsf{is\_local}\,(\mathsf{local}\,\mathcal{F})$$

To summarize, I have just established that any local formula supports applications of the frame rule. Similarly, I have proved in Coq that local formulae support applications of the rule of consequence and of rule of garbage collection. For the sake of readability, those results are presented as inference rules.

$$\frac{\mathsf{is\_local}\,\mathcal{F} \qquad \mathcal{F}\,H\,Q}{\mathcal{F}\,(H * H')\,(Q \star H')}\ \text{FRAME} \qquad\qquad \frac{\mathsf{is\_local}\,\mathcal{F} \qquad \mathcal{F}\,H\,(Q \star H'')}{\mathcal{F}\,(H * H')\,Q}\ \text{GC}$$

$$\frac{\mathsf{is\_local}\,\mathcal{F} \qquad H \rhd H' \qquad \mathcal{F}\,H'\,Q' \qquad Q' \blacktriangleright Q}{\mathcal{F}\,H\,Q}\ \text{CONSEQUENCE}$$

More generally, local formulae admit the following general elimination rule.

$$\frac{\mathsf{is\_local}\,\mathcal{F} \qquad H \rhd H_i * H_k \qquad \mathcal{F}\,H_i\,Q_f \qquad Q_f \star H_k \blacktriangleright Q \star H_g}{\mathcal{F}\,H\,Q}\ \text{COMBINED}$$

5.3.4 Extraction of invariants from pre-conditions

Another crucial property of local formulae is the ability to extract propositions and existentially-quantified variables from pre-conditions. To understand when such extractions are involved, consider the expression "$\mathsf{let}\,a = \mathsf{ref}\,2\,\mathsf{in}\,\mathsf{get}\,a$", and assume we have specified the term "$\mathsf{ref}\,2$" by saying that it returns a fresh location pointing towards an even integer, that is, through the post-condition "$\lambda a.\,\exists n.\,a \hookrightarrow n * [\mathsf{even}\,n]$". Now, to reason on the term "$\mathsf{get}\,a$", we need to prove the following proposition, where Q is some post-condition.

$$\forall a.\ [\![\mathsf{get}\,a]\!]\,(\exists n.\,a \hookrightarrow n * [\mathsf{even}\,n])\,Q$$

In order to read at the location a, we need a pre-condition of the form $a \hookrightarrow n$. So, we need to extract the existential quantification on n and the invariant "$\mathsf{even}\,n$", so as to change the goal to:

$$\forall a.\ \forall n.\ \mathsf{even}\,n \ \Rightarrow\ [\![\mathsf{get}\,a]\!]\,(a \hookrightarrow n)\,Q$$

Extrusion of the existentially-quantified variables and of propositions is precisely the purpose of the two following extraction lemmas[3], which are derivable from the definition of the predicate local. Similar extraction lemmas have appeared in previous work on Separation Logic (e.g., [2]).

EXTRACT-PROP
$$\frac{\mathsf{is\_local}\,\mathcal{F} \qquad (\mathcal{P} \Rightarrow \mathcal{F}\,H\,Q)}{\mathcal{F}\,([\mathcal{P}] * H)\,Q}$$

EXTRACT-EXISTS
$$\frac{\mathsf{is\_local}\,\mathcal{F} \qquad (\forall x.\ \mathcal{F}\,(H' * H)\,Q)}{\mathcal{F}\,((\exists x.\,H') * H)\,Q}$$

The flawed predicate local' defined earlier on does not involve a quantification on the heap representation h that satisfies the pre-condition H. This predicate local' allows deriving the frame rule, the rule of consequence and the rules of garbage collection, however it does not allow proving an extraction result such as EXTRACT-EXISTS. Intuitively, there is a problem related to the commutation of an existential quantifiers with a universal quantifier. In the definition local', the universal quantification on the input heap h is implicitly contained in the proposition $H \rhd H_i * H_k$, so the quantification on h comes after the existential quantification on H_i. On the contrary, in the definition of local, the universal quantification of h comes before the existential quantification on H_i.

5.4 Specification of imperative functions

I now focus on the specification of functions, defining the predicates $\mathsf{AppReturns}$ and Spec, and then generalizing those predicates to curried n-ary functions. Note: through the rest of this chapter, I write Coq values in lowercase and no longer in uppercase, for the sake of readability.

5.4.1 Definition of the predicates AppReturns and Spec

In an imperative setting, the evaluation formula takes the form $\mathsf{AppEval}\,f\,v\,h\,v'\,h'$, asserting that the application of a Caml function whose decoding is f to a value whose decoding is v in a store represented as h terminates and returns a value decoded as v' in a store represented as h'. Note that the arguments h and h' of $\mathsf{AppEval}$ here describe the entire memory store in which the evaluation of the application of f to v takes place, and not just the piece of memory involved for reasoning on that application. Here again, $\mathsf{AppEval}$ is an axiom upon which the CFML library is built. Its type is as follows.

$$\mathsf{AppEval} \quad : \quad \forall A\,B.\ \mathsf{Func} \to A \to \mathsf{Heap} \to B \to \mathsf{Heap} \to \mathsf{Prop}$$

[3]The Coq statement of the rule EXTRACT-EXISTS is as follows, where the definition of hexists is that given in a footnote in §5.2.2.

$$\mathsf{is\_local}\,\mathcal{F} \Rightarrow (\forall x.\ \mathcal{F}\,((J\,x) * H)\,Q) \Rightarrow \mathcal{F}\,((\mathsf{hexists}J) * H)\,Q$$

The definition of $\mathsf{AppReturns}_1$ in terms of $\mathsf{AppEval}$ is slightly more involved for imperative programs than for purely-functional ones, mainly because of the need to quantify over the piece of heap that is framed out during the reasoning on a function application, and of the need to take into account the fact that pieces of heap might be discarded after the execution of a function. Recall the type of $\mathsf{AppReturns}_1$, which is as follows.

$$\mathsf{AppReturns}_1 \quad : \quad \forall A\, B.\, \mathsf{Func} \to A \to \mathsf{Hprop} \to (B \to \mathsf{Hprop}) \to \mathsf{Prop}$$

The proposition "$\mathsf{AppReturns}_1\, f\, v\, H\, Q$" states that, if the input heap can be decomposed as the disjoint union of a heap h_i that satisfies the pre-condition H and of another heap h_k, then the application of f to v returns a value v' in a output heap that can be decomposed in three disjoint parts h_f, h_k and h_g such that $Q\, v'\, h_f$ holds. The heap h_k describes the piece of store that is being framed out and the heap h_g describes the piece of heap being discarded. In the formal definition that appears next, "$h_f \perp h_k \perp h_g$" denotes the pairwise disjointedness of the three heaps h_f, h_k and h_g.

$$\begin{array}{l}\mathsf{AppReturns}_1\, f\, v\, H\, Q \;\equiv \\ \quad \forall h_i\, h_k.\, \left\{\begin{array}{l} h_i \perp h_k \\ H\, h_i \end{array}\right. \;\Rightarrow\; \exists v'\, h_f\, h_g.\, \left\{\begin{array}{l} h_f \perp h_k \perp h_g \\ \mathsf{AppEval}\, f\, v\, (h_i + h_k)\, v'\, (h_f + h_k + h_g) \\ Q\, v'\, h_f \end{array}\right.\end{array}$$

Note that this definition is formalized in Coq, since only the predicate $\mathsf{AppEval}$ is taken as axiom in the CFML library.

A central result is that the predicate "$\mathsf{AppReturns}_1\, f\, v$" is a local formula, for any f and v.

$$\forall f\, v.\;\; \mathsf{is\_local}\,(\mathsf{AppReturns}_1\, f\, v)$$

In particular, this result implies that $\mathsf{AppReturns}_1$ is compatible with the frame rule, in the sense that the proposition $\mathsf{AppReturns}_1\, f\, v\, H_1\, Q_1$ implies the proposition $\mathsf{AppReturns}_1\, f\, v\, (H_1 * H_2)\, (Q_1 \star H_2)$.

The definition of Spec_1 in terms of $\mathsf{AppReturns}_1$ is very similar to the one involved in a purely-functional setting.

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

The main difference is the type of specifications. Here, the specification K takes the form $A \to \widetilde{B} \to \mathsf{Prop}$, where the shortand $\widetilde{B}$ is defined as follows.

$$\widetilde{B} \quad \equiv \quad \mathsf{Hprop} \to (B \to \mathsf{Hprop}) \to \mathsf{Prop}$$

The definition of $\mathsf{is\_spec}_1\, K$ asserts that, for any argument x, the predicate $K\, x$ is covariant. Covariance is captured by a generalized version of the predicate $\mathsf{Weakenable}$, shown next.

$$\mathsf{Weakenable}\, J \quad \equiv \quad \forall R\, R'.\, J\, R \Rightarrow (\forall H Q.\, R\, H\, Q \Rightarrow R'\, H\, Q) \Rightarrow J\, R'$$

5.4.2 Treatment of n-ary applications

The next step consists in defining $\mathsf{AppReturns}_n$ and Spec_n. As suggested earlier on, the treatment of curried functions in an imperative setting is trickier than in a purely-functional setting because every partial application might induce a side-effect.

Consider a program that contains an application of the form "$f\,x\,y$". From only looking at this piece of code, we do not know whether the evaluation of the partial application "$f\,x$" modifies the store. So, we have to assume that it might do so. One way to deal with curried applications is to name every intermediate result with a let-binding during the normalization process, e.g., changing "$f\,x\,y$" into "$\mathsf{let}\,g = f\,x\,\mathsf{in}\,g\,y$". However, this approach would not be practical at all. Instead, I wanted to preserve the nice and simple rule that was devised for pure programs, where $[\![f\,x_1 \ldots x_n]\!]$ is defined as "$\mathsf{AppReturns}_n\,f\,x_1 \ldots x_n$".

A simple and effective way to find appropriate definition of $\mathsf{AppReturns}_n$ in an imperative setting is to exploit the fact that a term "$f\,x\,y$" admits exactly the same behavior as the term "$\mathsf{let}\,g = f\,x\,\mathsf{in}\,g\,y$". Indeed, since characteristic formulae are sound and complete descriptions of program behaviors, the characteristic formulae of two terms that admit the same behavior must be logically equivalent. So, the characteristic formula of "$f\,x\,y$", which is "$\mathsf{AppReturns}_2\,f\,x\,y$", should be logically equivalent to the characteristic formula of "$\mathsf{let}\,g = f\,x\,\mathsf{in}\,g\,y$".

The characteristic formula of "$\mathsf{let}\,g = f\,x\,\mathsf{in}\,g\,y$", shown next, is a local formula that states that one needs to find an intermediate post-condition Q' for the application of f to x such that, for any g, the predicate "$Q'\,g$" is an appropriate pre-condition for the application of g to y.

$$\mathsf{local}\left(\lambda HQ.\ \exists Q'.\left\{\begin{array}{l}\mathsf{local}\,(\mathsf{AppReturns}_1\,f\,x)\,H\,Q'\\ \forall g.\ \mathsf{local}\,(\mathsf{AppReturns}_1\,g\,y)\,(Q'\,g)\,Q\end{array}\right.\right)$$

Since a predicate of the form "$\mathsf{AppReturns}_1\,f\,v$" is already a local formula, it does not change the meaning of the above formula to remove the applications of the predicate local that occur in front of $\mathsf{AppReturns}_1$. What then remains is a suitable definition for the predicate "$\mathsf{AppReturns}_2\,f\,x\,y$".

This analysis suggests the following formal definition for $\mathsf{AppReturns}_n$.

$$\begin{array}{l}\mathsf{AppReturns}_n\,f\,x_1 \ldots x_n\ \equiv\\ \quad\mathsf{local}\left(\lambda HQ.\ \exists Q'.\left\{\begin{array}{l}\mathsf{AppReturns}_1\,f\,x_1\,H\,Q'\\ \forall g.\ \mathsf{AppReturns}_{n-1}\,g\,x_2 \ldots x_n\,(Q'\,g)\,Q\end{array}\right.\right)\end{array}$$

Note that, by construction, a formula of the form $\mathsf{AppReturns}_n\,f\,v_1 \ldots v_n$ is always a local formula.

5.4.3 Specification of n-ary functions

The predicate Spec_n captures the specification of n-ary functions that are syntactically of the form $\lambda x_1 \ldots x_n.\,t$. Such functions do not perform any side effects on partial applications. Remark: a function of n arguments that is not syntactically

$$\begin{array}{lcl}
\mathsf{is_spec}_1\, K & \equiv & \forall x.\, \mathsf{Weakenable}\,(K\,x) \\
\mathsf{is_spec}_n\, K & \equiv & \forall x.\, \mathsf{is_spec}_{n-1}\,(K\,x) \\
\mathsf{Spec}_1\, f\, K & \equiv & \mathsf{is_spec}_1\, K \,\wedge\, \forall x.\, K\,x\,(\mathsf{AppReturns}_1\, f\, x) \\
\mathsf{Spec}_n\, f\, K & \equiv & \mathsf{is_spec}_n\, K \,\wedge\, \forall x.\, \mathsf{AppPure}\, f\, x\, (\lambda g.\, \mathsf{Spec}_{n-1}\, g\, (K\,x))
\end{array}$$

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

Figure 5.4: Formal definition of the imperative version of Spec_n

of the form $\lambda x_1 \ldots x_n.t$ can be specified in terms of $\mathsf{AppReturns}_n$ but cannot be specified using the predicate Spec_n.

To express that the application of a function f to an argument x is pure, we could try to define $\mathsf{Spec}_2\, f\, K$ as "$\forall x.\ \mathsf{AppReturns}_1\, f\, x\, [\,]\, (\lambda g.\, [\mathsf{Spec}_1\, g\, (K\,x)])$". However, this definition does not work out because it does not allow proving the introduction lemma and the elimination lemma for Spec_n, which capture the following equivalence (side conditions are omitted).

$$\mathsf{Spec}_n\, f\, K \quad \Longleftrightarrow \quad \big(\forall x_1 \ldots x_n.\ K\, x_1 \ldots x_n\, (\mathsf{AppReturns}_n\, f\, x_1 \ldots x_n)\big)$$

To prove this result, we would need to know that the partial application of a function f to an argument x always returns the same function g. Yet, a predicate form $\mathsf{AppReturns}_1\, f\, x\, [\,]\, (\lambda g.\, [P\, g])$ does not capture this property, because the exact addresses of the locations allocated during the evaluation of the application of f to x may depend on the addresses allocated in the piece of heap that has been framed out.[4] Intuitively, the problem is that the proposition $\mathsf{AppReturns}_1\, f\, x\, [\,]\, (\lambda g.\, [P\, g])$ does not disallow side-effects, whereas the development of introduction and elimination lemmas for curried n-ary functions requires the knowledge that partial applications do not involve side-effects. To work around this problem, I introduce a more precise predicate, called $\mathsf{AppPure}$, for describing applications that are completely pure.

The proposition $\mathsf{AppPure}\, f\, v\, P$ asserts that the application of f to v returns a value v' satisfying the predicate P, without reading, writing, nor allocating in the store. The predicate $\mathsf{AppPure}$ is defined in terms of $\mathsf{AppEval}$, using a proposition of the form "$\mathsf{AppEval}\, f\, v\, h\, v'\, h$", where the output heap h is exactly the same as the input heap h.

$$\mathsf{AppPure}\, f\, v\, P \quad \equiv \quad \exists v'.\ P\, v' \ \wedge\ (\forall h.\, \mathsf{AppEval}\, f\, v\, h\, v'\, h)$$

Note that the result v' produced by the evaluation of "$f\, v$" does not depend on the input heap h, which is universally-quantified after the existential quantification of v'.

[4] To illustrate the argument, consider the function $\lambda x.\, \mathsf{let}\, l = \mathsf{ref}\, 3\, \mathsf{in}\, \lambda y.\, l$, which allocates a memory cell and then returns a constant function that always returns the allocated location. This function satisfies the property $\mathsf{AppReturns}_1\, f\, x\, [\,]\, (\lambda g.\, [\mathsf{Spec}_1\, g\, (\lambda y\, R.\, \mathsf{True})])$ for any argument x. However, two successive applications of f to x return two different functions.

The predicate Spec_2 can now be defined in terms of $\mathsf{AppPure}$ and Spec_1. Compared with the definition of Spec_2 from the purely-functional setting, the only difference is the replacement of the predicate $\mathsf{AppReturns}$ with the predicate $\mathsf{AppPure}$.

$$\mathsf{Spec}_2\, f\, K \quad \equiv \quad \mathsf{is\_spec}_2\, K \;\wedge\; \forall x.\; \mathsf{AppPure}\, f\, x\, (\lambda g.\, \mathsf{Spec}_1\, g\, (K\, x))$$

The general definitions of Spec_n and of $\mathsf{is\_spec}_n$ appear in Figure 5.4.

5.5 Characteristic formulae for imperative programs

5.5.1 Construction of characteristic formulae

The algorithm for constructing characteristic formulae for imperative programs appears in Figure 5.5, and are explained next. For the sake of clarity, contexts and decoding of values are left implicit. I have set up a notation layer for pretty-printing characteristic formulae for imperative programs, in a very similar way as described in the previous chapter for purely-functional programs. I omit the details here.

As explained in the introduction, an application of the predicate local is introduced at every node of a characteristic formula. A value v admits a pre-condition H and a post-condition Q if the current heap, which by assumption satisfies the predicate H, also satisfies the predicate $Q\, v$. This entailment is written $H \rhd Q\, v$. The application of a n-ary function is described through the predicate $\mathsf{AppReturns}_n$. Here, the application of local in front of $\mathsf{AppReturns}_n$ is redundant, yet I leave it for the sake of uniformity. The treatment of crash, of conditionals and of function definitions is quite similar to the treatment given for the purely-functional setting. In short, it suffices to replace occurrences of a post-condition P with a pre-condition H and a post-condition Q. In practice, it is useful to consider a direct construction for the characteristic formulae of terms of the form "$\mathsf{if}\, t_0\, \mathsf{then}\, t_1\, \mathsf{else}\, t_2$".

$$\begin{array}{l}[\![\mathsf{if}\, t_0\, \mathsf{then}\, t_1\, \mathsf{else}\, t_2]\!] \;\equiv \\ \quad \mathsf{local}\, (\lambda H Q.\; \exists Q'.\; [\![t_0]\!]\, H\, Q' \;\wedge\; [\![t_1]\!]\, (Q'\, \mathsf{true})\, Q \;\wedge\; [\![t_2]\!]\, (Q'\, \mathsf{false})\, Q)\end{array}$$

One can prove that this direct formula is logically equivalent to the characteristic formula of the term "$\mathsf{let}\, x = t_0\, \mathsf{in}\, \mathsf{if}\, x\, \mathsf{then}\, t_1\, \mathsf{else}\, t_2$".

The treatment of let-bindings has already been described in the introduction. In a term "$\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2$", the post-condition Q' of t_1 is quantified existentially, and then $Q'\, x$ describes the pre-condition for t_2. Sequences are a particular case of let-bindings, where the result of t_1 is of type unit and thus need not be named.

For a polymorphic let-binding, the bound term must be a syntactic value, due to the value restriction. Consider the typed term "$\mathsf{let}\, x = \hat{w}_1\, \mathsf{in}\, \hat{t}_2$", where $\hat{w}_1$ stands for a possibly-polymorphic value. The construction of the corresponding characteristic formula is formally described as follows.

$$[\![\mathsf{let}\, x = \hat{w}_1\, \mathsf{in}\, \hat{t}_2]\!]^{\Gamma} \quad \equiv \quad \mathsf{local}\, (\lambda H Q.\; \forall X.\; X = \lceil \hat{w}_1 \rceil \;\Rightarrow\; [\![\hat{t}_2]\!]^{(\Gamma, x \mapsto X)}\, H\, Q)$$

If S denotes the type of $\hat{w}_1$, then the formula universally quantifies over a value X of type $[\![S]\!]$, and provides the assumption "$X = \lceil \hat{w}_1 \rceil$", which asserts that the logical

$$\llbracket v \rrbracket \equiv \mathsf{local}\,(\lambda HQ.\ H \rhd Q\,v)$$

$$\llbracket f\,v_1 \ldots v_n \rrbracket \equiv \mathsf{local}\,(\lambda HQ.\ \mathsf{AppReturns}_n\,f\,v_1 \ldots v_n\,H\,Q)$$

$$\llbracket \mathsf{crash} \rrbracket \equiv \mathsf{local}\,(\lambda HQ.\ \mathsf{False})$$

$$\llbracket \mathsf{if}\,v\,\mathsf{then}\,t_1\,\mathsf{else}\,t_2 \rrbracket \equiv \mathsf{local}\,(\lambda HQ.\ (v = \mathsf{true} \Rightarrow \llbracket t_1 \rrbracket\,H\,Q) \wedge (v = \mathsf{false} \Rightarrow \llbracket t_2 \rrbracket\,H\,Q))$$

$$\llbracket \mathsf{let}\,x = t_1\,\mathsf{in}\,t_2 \rrbracket \equiv \mathsf{local}\,(\lambda HQ.\ \exists Q'.\ \llbracket t_1 \rrbracket\,H\,Q' \wedge \forall x.\ \llbracket t_2 \rrbracket\,(Q'\,x)\,Q)$$

$$\llbracket t_1\,;\,t_2 \rrbracket \equiv \mathsf{local}\,(\lambda HQ.\ \exists Q'.\ \llbracket t_1 \rrbracket\,H\,Q' \wedge \llbracket t_2 \rrbracket\,(Q'\,tt)\,Q)$$

$$\llbracket \mathsf{let}\,x = w_1\,\mathsf{in}\,t_2 \rrbracket \equiv \mathsf{local}\,(\lambda HQ.\ \forall x.\ x = w_1 \Rightarrow \llbracket t_2 \rrbracket\,H\,Q)$$

$$\llbracket \mathsf{let\,rec}\,f = \lambda x_1 \ldots x_n.\,t_1\,\mathsf{in}\,t_2 \rrbracket \equiv \mathsf{local}\,(\lambda HQ.\ \forall f.\ \mathcal{H} \Rightarrow \llbracket t_2 \rrbracket\,H\,Q)$$
$$\text{with } \mathcal{H} \equiv \forall K.\,\mathsf{is\_spec}_n\,K \wedge (\forall x_1 \ldots x_n.\ K\,x_1 \ldots x_n\,\llbracket t_1 \rrbracket) \Rightarrow \mathsf{Spec}_n\,f\,K$$

$$\llbracket \mathsf{while}\,t_1\,\mathsf{do}\,t_2 \rrbracket \equiv \mathsf{local}\,(\lambda HQ.\ \forall R.\,\mathsf{is\_local}\,R \wedge \mathcal{H} \Rightarrow R\,H\,Q)$$
$$\text{with } \mathcal{H} \equiv \forall H'Q'.\ \llbracket \mathsf{if}\,t_1\,\mathsf{then}\,(t_2\,;\,|R|)\,\mathsf{else}\,tt \rrbracket\,H'\,Q' \Rightarrow R\,H'\,Q'$$

$$\llbracket \mathsf{for}\,i = a\,\mathsf{to}\,b\,\mathsf{do}\,t \rrbracket \equiv \mathsf{local}\,(\lambda HQ.\ \forall S.\,\mathsf{is\_local}_1\,S \wedge \mathcal{H} \Rightarrow S\,a\,H\,Q)$$
$$\text{with } \mathcal{H} \equiv \forall iH'Q'.\ \llbracket \mathsf{if}\,i \le b\,\mathsf{then}\,(t\,;\,|S\,(i+1)|)\,\mathsf{else}\,tt \rrbracket\,H'\,Q' \Rightarrow S\,i\,H'\,Q'$$

Figure 5.5: Characteristic formula generator for imperative programs

variable X corresponds to the program value $\hat{w}_1$. When S is a polymorphic type, the equality "$X = \lceil \hat{w}_1 \rceil$" relates two functions from the logic (recall that I assume the target logic to include functional extensionality). For example, if $\hat{w}_1$ is the value nil (the empty Caml list), then X is equal to Nil (the empty Coq list), where both X and Nil admit the Coq type "$\forall A.\ \mathsf{list} A$".

The characteristic formulae of while-loops and of for-loops are stated using an anti-quotation operator, written $|R|$, used to refer to Coq predicates inside the computation of the characteristic formula of a term. This operator is such that $[\![|R|]\!]$ is equal to R. Recall that "$\mathsf{is\_local}\, R$" asserts that R is a local predicate and that "$\mathsf{is\_local}_1\, S$" asserts that "$S\, i$" is a local predicate for every argument i.

Primitive functions for manipulating references have been explained in Chapter 3. I recall their specification, and give the specification of the function cmp that compares two pointers.

$$
\begin{array}{lll}
\forall A. & \mathsf{Spec}_1\,\mathsf{ref} & (\lambda v\, R.\quad R\,[\,]\,(\lambda l.\, l \hookrightarrow_A v)) \\
\forall A. & \mathsf{Spec}_1\,\mathsf{get} & (\lambda l\, R.\quad \forall v.\ R\,(l \hookrightarrow_A v)\,(\backslash{=}\, v \star (l \hookrightarrow_A v))) \\
\forall A. & \mathsf{Spec}_2\,\mathsf{set} & (\lambda l\, v\, R.\ \forall v'.\ R\,(l \hookrightarrow_A v')\,(\#\,(l \hookrightarrow_A v))) \\
 & \mathsf{Spec}_2\,\mathsf{cmp} & (\lambda l\, l'\, R.\ R\,[\,]\,(\lambda b.\, [b = \mathsf{true} \Leftrightarrow l = l'])) \\
\end{array}
$$

5.5.2 Generated axioms for top-level definitions

The current implementation only supports top-level definitions of values and functions. It does not yet support general top-level declaration of the form "$\mathsf{let}\, x = t$", but this should be added soon. For the time being, CFML can be used to verify imperative functions and, in particular, imperative data structures.

5.6 Extensions

The treatment of mutually-recursive functions and of pattern-matching developed in the previous chapter can be immediately adapted to an imperative setting. It suffices to replace every application of a formula to a post-condition P with the application of the same formula to a pre-condition H and to a post-condition Q. However, the treatment of assertions in an imperative setting is different because assertions might perform side effects. Some systems require assertions not to perform any side effects, nevertheless it can be useful to allow assertions to use local effects, i.e. effects that are used to evaluate the assertion but that are not observable by the rest of the program. In this section, I describe a treatment of assertions that ensures that programs remain correct regardless of whether assertions are executed or not. I then explain how to support null pointers and strong updates.

5.6.1 Assertions

In an imperative setting, an assertion expression takes the form "assert t". If t returns the boolean value true, then the term "assert t" returns the unit value. Otherwise,

if t returns false, then the program crashes. In general, assertions are allowed to access and even modify the store, as asserted by the reduction rule for assertions.

$$\frac{t_{/m} \Downarrow \mathsf{true}_{/m'}}{(\mathsf{assert}\ t)_{/m} \Downarrow \mathit{tt}_{/m'}}$$

Let me first give a characteristic formula that corresponds closely to this reduction rule. Let H be a pre-condition and Q be a post-condition for the term "$\mathsf{assert}\ t$". The term t is evaluated in a heap satisfying H. It must return a value equal to the boolean true, that is, satisfying the predicate $(= \mathsf{true})$. The output heap should satisfy the predicate "$Q\ \mathit{tt}$". So, the post-condition for t is $\backslash{=}\,\mathsf{true} \star (Q\ \mathit{tt})$. To summarize, the characteristic formulae associated with an assertion may be built as follows.

$$[\![\mathsf{assert}\ t]\!] \quad \equiv \quad \lambda HQ.\ [\![t]\!]\, H\, (\backslash{=}\,\mathsf{true} \star (Q\ \mathit{tt}))$$

The above formula only asserts that the program is correct when assertions are executed. Yet, most programmers expect their code to be correct regardless of whether assertions are executed or not. I next explain how to build a characteristic formula for assertions that properly captures the irrelevance of assertions, while still allowing assertions to access and modify the store, as this possibility can be useful in general.

To that end, I impose that the execution of the assertion "$\mathsf{assert}\ t$" produces an output heap that satisfies the same invariant as the input heap it receives. If H denotes the pre-condition of the term t, then the post-condition of t is $\backslash{=}\,\mathsf{true} \star H$. The post-condition Q of the term "$\mathsf{assert}\ t$" describes the same heap as H, so the predicate H should entail the predicate "$Q\ \mathit{tt}$". The appropriate characteristic formula for assertions is thus as follows.

$$[\![\mathsf{assert}\ t]\!] \quad \equiv \quad \lambda HQ.\ [\![t]\!]\, H\, (\backslash{=}\,\mathsf{true} \star H) \ \wedge\ H \rhd Q\, tt$$

With such a characteristic formula, although an assertion cannot break the invariants satisfied by the heap, it may modify values stored in the heap. For example, the expression "$\mathsf{assert}\ (\mathsf{set}\, x\, 4\,;\, \mathsf{true})$" updates the heap at location x, but it does not break an invariant asserting that the location x contains an even integer, e.g. "$\exists n.\, x \hookrightarrow n * [\mathsf{even}\ n]$". So, technically, the final result computed by a program may depend on whether assertions are executed or not. However, this result always satisfies the specification of the program, regardless of whether assertions are executed or not.

5.6.2 Null pointers and strong updates

Motivation The ML type system guarantees type soundness: a well-typed program can never crash or get stuck. In particular, if a program involves a location l of type $\mathsf{ref}\,\tau$, then the store indeed contains a value of type τ at location l. To achieve this soundness result, ML gives up on at least two popular features from C-like languages: null pointers and strong updates. Null pointers are typically used to encode

empty data structures. In Caml, an explicit option type is generally used to translate C programs using null pointers, and this encoding has a small but measurable cost in terms of execution speed and memory consumption. Strong updates allow reusing a same memory location at several types, which is again a useful feature for saving some memory space. Strong updates are also convenient for initializing cyclic data structures. In this section, I explain how to recover null pointers and strong update in Caml while still being able to prove programs correct using characteristic formulae.

Null pointers To support null pointers, I introduce a constant location null in the programming language, implemented as the memory address 0. I also introduce in the logic a corresponding constant of type Loc, called Null. The decoding operation for locations is such that $\lceil \mathsf{null} \rceil$ is equal to Null.

A singleton heap predicate of the form "$l \hookrightarrow_{\mathcal{T}} V$" should imply that l is not a null pointer. To that end, I update the definition of the singleton heap predicate with an assumption $l \neq \mathsf{Null}$, as follows.

$$l \hookrightarrow_{\mathcal{T}} V \quad \equiv \quad \lambda h.\ h = (l \rightarrow_{\mathcal{T}} V) \wedge l \neq \mathsf{Null}$$

With this new definition, the specification of ref, recalled next, ensures that the location l being returned by a call to ref is distinct from the null location.

$$\forall A. \quad \mathsf{Spec}_1\, \mathsf{ref}\, (\lambda v\, R.\ R\, [\,]\, (\lambda l.\, l \hookrightarrow_A v))$$

The pointer comparison function cmp can be used to test at runtime whether a given pointer is null. For example, one can define a function is_null that expects a location l and returns a boolean b that is true if and only if the location l is the null location. The specification of is_null appears below.

$$\mathsf{Spec}_1\, \mathsf{is\_null}\, (\lambda l\, R.\ R\, [\,]\, (\lambda b.\, [b = \mathsf{true} \Leftrightarrow l = \mathsf{Null}]))$$

Strong updates Characteristic formulae accommodate strong updates quite naturally because the type of memory cells is not carried by the type of pointers, since all pointers admit the constant type loc in weak-ML. Instead, the type of the contents of a memory cell appears in a heap predicate of the form $l \hookrightarrow_{\mathcal{T}} V$, which asserts that the location l contains a Caml value that corresponds to the Coq value V of type $\mathcal{T}$. Remark: earlier work on the logic of Bunched Implication [42], a precursor of Separation Logic [77], has pointed out the fact that working with heap predicates allows reasoning on strong updates [9]. This possibility was exploited in Hoare Type Theory [61], which builds upon Separation Logic.

To support strong updates, it therefore suffices to generalize the specification of the primitive function set, allowing the type A of the argument to differ from the type A' of the previous contents of the location. This generalized specification is formally stated as follows.

$$\forall A. \quad \mathsf{Spec}_2\, \mathsf{set}\, (\lambda\, l\, v\, R.\ \forall A'.\, \forall (v' : A').\, R\, (l \hookrightarrow_{A'} v')\, (\#\, (l \hookrightarrow_A v)))$$

When the type of memory cells is allowed to evolve through the execution of a program, it is generally useful to be able to cast a pointer from a type to another. A subtyping rule of the form "$\mathsf{ref}\,\tau \leq \mathsf{ref}\,\tau'$" would here make sense. This subtyping rule is obviously unsound in ML, but it does not break the soundness of characteristic formulae because both the type $\mathsf{ref}\,\tau$ and the type $\mathsf{ref}\,\tau'$ are reflected in the logic as the type Loc. To encode the subtyping rule in Caml, I introduce a coercion function called cast, of type "$\mathsf{ref}\,\tau \to \mathsf{ref}\,\tau'$", which behaves like the identity function. Applications of the function cast are eliminated on-the-fly by the CFML generator, immediately after type-checking.

Implementation in Caml Null references and strong references can be implemented in Caml with the help of the primitive function Obj.magic, of type `'a -> 'b`. This function allows fooling the type system by arbitrarily changing the type of expressions. Remark: this encoding of strong references and null pointers in Caml is a standard trick.

Figure 5.6 contains the signature and the implementation of a library for C-style manipulation of pointers. The module includes functions sref and sget and sset for manipulating a Caml reference at a different type than that carried by the pointer. It also includes the function cmp for comparing pointers of different types, the function cast for changing the type of a pointer, the constant null which denotes the null reference, and the function is_null which is a specialization of the comparison function cmp for testing whether a given pointer is null.

Remark: it is also possible to build a library where all pointers admit a constant type called sref. For example, in this setting, the function ref admits the type $\forall A.\, A \to \mathsf{sref}$ and the function get admits the type $\forall A.\, \mathsf{sref} \to A$. This alternative presentation can be more convenient in some particular developments, however it seems that, in general, the use of a constant type for pointers requires a greater number of type annotation in source code than the use of type-carrying pointers.

To conclude, characteristic formulae allow for a safe and practical integration of advanced pointer manipulations in a high-level programming language like Caml.

5.7 Additional tactics for the imperative setting

In this section, I describe the tactics for manipulating characteristic formulae that are specific to the imperative setting. I start with a core tactic that helps proving heap entailment relations. I then explain how the frame rule is automatically applied by the tactic that handles reasoning about applications. Finally, I give a brief overview of the other tactics involved.

5.7.1 Tactic for heap entailment

The tactic `hsimpl` helps proving goals of the form $H_1 \rhd H_2$. It works modulo associativity and commutativity of the separating conjunction, and it is able to instantiate

```
module type PointerSig = sig
   val sref : 'b -> 'a ref
   val sget : 'a ref -> 'b
   val sset : 'a ref -> 'b -> unit
   val cmp : 'a ref -> 'b ref -> bool
   val cast : 'a ref -> 'b ref
   val null : 'a ref
   val is_null : 'a ref -> bool
end

module Pointer : PointerSig = struct
   let sref x = magic (ref x)
   let sget p = !(magic p)
   let sset p x = (magic p) := x
   let cmp p1 p2 = ((magic p1) == p2)
   let cast p = magic p
   let null = magic (ref ())
   let is_null p = cmp (magic null) p
end
```

Figure 5.6: Signature and implementation of advanced pointer manipulations

the existential quantifiers occurring in H_2 by exploiting information available in H_1. For example, consider the following goal, in which `?V` and `?H` denote Coq unification variables.

```
    (x ~> T X) \* (l ~> Mlist T L) \* (h ~~> y)
==> (Hexists L', l ~> Mlist L') \* (h ~~> ?V) \* [y = ?V] \* (?H)
```

The tactic `hsimpl` solves this goal as follows. First, it introduces a Coq unification variable `?L'` in place of the existentially-quantified variable `L'`. (Alternatively, one may explicitly provide an explicit witness for `L'` as argument of `hsimpl`.) Second, the tactic tries to cancel out heap predicates from the left-hand side with those from the right-hand side. This process unifies `?L'` with `L` and `?V` with `y`. The embedded proposition `[y = ?V]` is extracted as a subgoal. Since `?V` has been instantiated as `y`, the subgoal is trivial to prove. After simplification, the remaining goal is `x~>T X ==> ?H`. At this point, `?H` is unified with the predicate `x~>T X`, and the goal is solved.

The tactic `hsimpl` expects the right hand-side of the goal to be free of existentials and of embedded propositions. The purpose of the tactic `hextract` is to set the goal in that form, by pulling existential quantifiers and embedded propositions out of the goal and putting them in the context. Documentation and examples can be found in the Coq development for additional details.

5.7.2 Automated application of the frame rule

The tactic `xapp` enables one to exploit the specification of a function for reasoning on an application of that function. It automatically applies the frame rule on the heap predicates that are not involved in the reasoning on the function application. The implementation of the tactic relies on a corollary of the elimination lemma for the predicate Spec, shown next.

$$\begin{cases} \mathsf{Spec}\, f\, K \\ \forall R.\ \mathsf{is\_local}\, R \Rightarrow K\, x\, R \Rightarrow R\, H\, Q \end{cases} \quad \Rightarrow \quad \mathsf{AppReturns}\, f\, x\, H\, Q$$

Let me illustrate the working of `xapp` on an example. Consider an application of the function incr to a pointer `x`, under a pre-condition asserting that `x` is bound to the value 3 in memory and that another pointer `y` is bound to the value 7. Assume that the post-condition is a unification variable called `?Q`. Note that post-conditions are generally reduced to a unification variable because we try to infer as much information as possible. The goal then takes the following form.

```
AppReturns incr x (x ~~> 3 \* y ~~> 7) ?Q
```

I am going to explain in detail how the tactic `xapp` exploits the specification of the function incr for discharging the goal and in the same time infer that the post-condition `?Q` should be instantiated as "`# (x ~~> 4) \* (y ~~> 7)`". Recall the specification of incr expressed in terms of the predicate Spec. It is stated next (not using the notation associated with Spec).

```
spec_1 incr (fun r R => forall n, R (r ~~> n) (# r ~~> n+1))
```

Applying the lemma mentioned earlier on to that specification leaves:

```
forall R, is_local R ->
  (forall n, R (x ~~> n) (# x ~~> n+1)) ->
  R (x ~~> 3 \* y ~~> 7) ?Q
```

At this point, the tactic `xapp` instantiates the hypothesis "`forall n, R (x ~~> n) (# x ~~> n+1)`" with unification variables. Let `?N` denote the Coq unification variable that instantiaties `n`. The hypothesis becomes as follows.

```
R (x ~~> ?N) (# x ~~> ?N+1)
```

The tactic `xapp` then exploits the hypothesis "`is_local R`" by applying the frame rule (technically, the frame rule combined with the rule of consequence) to the conclusion "`R (x ~~> 3 \* y ~~> 7) ?Q`". This application leaves three subgoals.

```
1)   R ?H1 ?Q1
2)   x ~~> 3 \* y ~~> 7 ==> ?H1 \* ?H2
3)   ?Q1 \*+ ?H2 ==> ?Q
```

The first goal is solved using the hypothesis "`R (x ~~> ?N) (# x ~~> ?N+1)`", instantiating `?H1` as `(x ~~> ?N)` and `?Q1` as `(# x ~~> ?N+1)`. Two subgoals remain.

```
2)   (x ~~> 3) \* (y ~~> 7) ==> (x ~~> ?N) \* ?H2
3)   (# x ~~> ?N+1) \*+ ?H2 ==> ?Q
```

The tactic `hsimpl` is invoked on goal (2). It unifies `?N` with the value `3` and `?H2` with `(y ~~> 7)`. Calling `hsimpl` on goal (3) then unifies the post-condition `?Q` with "`(# x ~~> 3+1) \*+ (y ~~> 7)`", which is equivalent to the expected instantiation of `?Q`.

To summarize, the tactic `xapp` is able to exploit a known specification as well as the information available in the pre-condition for inferring the post-condition of a function application. In particular, the application of the frame rule is entirely automated. In the example presented above, the instantiation of the ghost variable `n` could be inferred. However, there are cases where ghost variables need to be explicitly provided. So, the tactic `xapp` accepts a list of arguments that can be exploited for instantiating ghost variables.

5.7.3 Other tactics specific to the imperative setting

All the tactics described in this section apply to a goal of the form $\mathcal{F}\,H\,Q$, where $\mathcal{F}$ is a predicate that satisfies the predicate is_local, and where H and Q denote a pre- and a post-condition, respectively. The predicate $\mathcal{F}$ is typically a characteristic formula, but it may also be a universally-quantified predicate coming from the characteristic formula of a loop or from the specification of a function such as iter.

The tactic `xseq` is a specialized version of `xlet` for reasoning on sequences. The tactics `xfor` and `xwhile` are used to reason on loops using the characteristic formulae that support local reasoning. The tactics `xfor_inv` and `xwhile_inv` apply a lemma for replacing the general form of the characteristic formula for a loop with a specialized characteristic formula based on an invariant. The loop invariant to be used may be directly provided as argument to those tactics.

The tactic `xextract` pulls the existential quantifiers and the embedded propositions out of a pre-condition. The tactic `xframe` allows applying the frame rule manually, which might be useful for example to reason on a local let-binding. The tactic `xchange` helps applying a focus or an unfocus lemma. The tactic takes as argument a partially-instantiated lemma whose conclusion is a heap entailment relation, and exploits it to rewrite the pre-condition. The tactic `xchange_post` performs a similar task on the post-condition. The tactic `xgc` enables discarding some heap predicates from the pre-condition, and the tactic `xgc_post` enables discarding heap predicates from the post-condition.

Chapter 6

Soundness and completeness

In this chapter, I prove that characteristic formulae for pure programs are sound and complete. The soundness theorem states that if one can prove that the characteristic formula of a term t holds of a post-condition P then the term t terminates and returns a value that satisfies P. Characteristic formulae are built from the source code in a compositional way and that they can even be displayed in a way that closely resemble source code, thus, intuitively, proving the soundness of a characteristic formula with respect to the source code it describes should be relatively direct. The syntactic soundness proof that I give is indeed quite simple.

The completeness theorem states that if a term t evaluates to a value v then the characteristic formula for t holds of the most-general specification for v. If the value v does not contain any first-class function, then its most-general specification is simply the predicate "being equal to v". The case where v contains functions is slightly more subtle, since characteristic formulae only allow specifying the extensional behavior of functions and do not enable stating properties about the source code of a function closure. In this chapter, I explain how to take that restriction into account in the statement of the completeness theorem.

The last part of this chapter is concerned with justifying the soundness of the treatment of polymorphism. In characteristic formulae, I quantify type variables over the sort Type, which corresponds to the set of all Coq types, instead of restricting the quantification to the set of Coq types that correspond to some weak-ML type. It is more convenient in practice to quantify over Type because it is the default sort in Coq. I explain in this chapter why the quantification over Type is correct.

6.1 Additional definitions and lemmas

6.1.1 Interpretation of Func

To justify the soundness of characteristic formulae, I give a concrete interpretation to the type Func and to the predicate AppEval. I interpret Func as the set of all well-typed function closures. Let is_well_typed_closure be a predicate that characterizes well-typed values of the form $\mu f.\Lambda\overline{A}.\lambda x.\hat{t}$. The type Func is then constructed as dependent pairs made of a value $\hat{v}$ and of a proof that $\hat{v}$ satisfies the predicate is_well_typed_closure.

$$\mathsf{Func} \quad \equiv \quad \Sigma_{\hat{v}}(\mathsf{is\_well\_typed\_closure}\,\hat{v})$$

Observe that Func is a type built upon the *syntax* of source code from the programming language. So, a Coq realization of Func would involve a deep embedding of the source language. This indirection through syntax avoids the circularity traditionally associated with models of higher-order stores, where heaps contain functions and functions are interpreted in terms of heaps.

To prove interesting properties about characteristic formulae, I need a decoder for function closures that are created at runtime. Recall that decoders are only applied to well-typed values. I define the decoding of a well-typed function closure as the function closure itself, viewed as a value of type Func. A well-typed function closure $\mu f.\Lambda\overline{A}.\lambda x.\hat{t}$ can indeed be viewed as a value of the type Func, because Func denotes the set of all well-typed function closures. The formal definition is as follows.

$$\lceil \mu f.\Lambda\overline{A}.\lambda x.\hat{t} \rceil^{\Gamma} \quad \equiv \quad (\mu f.\Lambda\overline{A}.\lambda x.\hat{t},\ \mathcal{H}) : \mathsf{Func}$$
$$\text{where } \mathcal{H} \text{ is a proof of ``}\mathsf{is\_well\_typed\_closure}\,\hat{v}\text{''}$$

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

6.1.2 Reciprocal of decoding: encoding

In the proofs, I rely on the fact that the decoding operator yields a bijection between the set of all well-typed Caml values and the set of all Coq values that admit a type of the form $\llbracket T \rrbracket$. To show that decoding is bijective, I give the inverse translation, called encoding. In this section, I describe the translation from (a subset of) Coq types into weak-ML types, written $\lfloor\!\lfloor \mathcal{T} \rfloor\!\rfloor$, and the translation of Coq values into typed program values, written $\lfloor V \rfloor$. (Recall that $\mathcal{T}$ denotes a Coq type of the form $\llbracket T \rrbracket$.)

The definition of the inverse translation $\lfloor\!\lfloor \cdot \rfloor\!\rfloor$ is as simple as that of the translation $\llbracket \cdot \rrbracket$. Note that $\lfloor\!\lfloor \cdot \rfloor\!\rfloor$ describes a partial function in the sense that not all Coq types are the image of some weak-ML type.

$$\begin{array}{lcl}
\lfloor\!\lfloor A \rfloor\!\rfloor & \equiv & A \\
\lfloor\!\lfloor \mathsf{Int} \rfloor\!\rfloor & \equiv & \mathsf{int} \\
\lfloor\!\lfloor C\,\overline{\mathcal{T}} \rfloor\!\rfloor & \equiv & C\,\lfloor\!\lfloor \overline{\mathcal{T}} \rfloor\!\rfloor \\
\lfloor\!\lfloor \mathsf{Func} \rfloor\!\rfloor & \equiv & \mathsf{func} \\
\lfloor\!\lfloor \forall\overline{A}.\mathcal{T} \rfloor\!\rfloor & \equiv & \forall\overline{A}.\ \lfloor\!\lfloor \mathcal{T} \rfloor\!\rfloor
\end{array}$$

The reciprocal of the decoding operation is called *encoding*. The encoding of a Coq value V of type $\mathcal{T}$, written $\lfloor V \rfloor$, produces a typed program value $\hat{v}$ of type $[\![\mathcal{T}]\!]$. In the definition of the encoding operator, shown below, values on the left-hand side are closed Coq values, given with their type, and values on the right-hand side are closed program values, which are annotated with weak-ML types.

$$\begin{array}{lll} \lfloor n & : \mathsf{Int} \rfloor & \equiv \quad n^{\mathsf{int}} \\ \lfloor D\,\overline{\mathcal{T}}(V_1,\ldots,V_n) & : C\,\overline{\mathcal{T}} \rfloor & \equiv \quad D\,[\![\overline{\mathcal{T}}]\!]\,(\lfloor V_1 \rfloor,\ldots,\lfloor V_n \rfloor) \\ \lfloor (\mu f.\Lambda\overline{A}.\lambda x.\hat{t},\ \mathcal{H}) & : \mathsf{Func} \rfloor & \equiv \quad \mu f.\Lambda\overline{A}.\lambda x.\hat{t} \\ \lfloor V & : \forall\overline{A}.\ T \rfloor & \equiv \quad \Lambda\overline{A}.\,\lfloor V\,\overline{A} \rfloor \end{array}$$

The encoding operation is presented here as a meta-level operation, whose behavior depends on the type of its argument. If this operation were to be defined in Coq, it would rather be presented as a family of operators $\lfloor V \rfloor$, indexed with the type of the argument V.[1]

By construction, $[\![\cdot]\!]$ is the inverse function of $\lceil\!\lceil \cdot \rceil\!\rceil$, defined in §4.3.4, and $\lfloor \cdot \rfloor$ is the inverse function of $\lceil \cdot \rceil$, defined in §4.3.5. Those results are formalized through the following lemma.

Lemma 6.1.1 (Inverse functions)

$$\begin{array}{lcll} [\![\ \lceil\!\lceil T \rceil\!\rceil\ ]\!] & = & T & \textit{where } T \textit{ is a weak-ML type} \\ \lceil\!\lceil\ [\![\mathcal{T}]\!]\ \rceil\!\rceil & = & \mathcal{T} & \textit{where } \mathcal{T} \textit{ is a Coq type of the form } \lceil\!\lceil T \rceil\!\rceil \\ \lfloor\ \lceil \hat{v} \rceil\ \rfloor & = & \hat{v} & \textit{where } \hat{v} \textit{ is a well-typed weak-ML value} \\ \lceil\ \lfloor V \rfloor\ \rceil & = & V & \textit{where } V \textit{ is Coq value with a type of the form } \lceil\!\lceil T \rceil\!\rceil \end{array}$$

Proof The treatment of functions and of polymorphism are not immediate.

First, consider a well-typed function closure $\hat{v}$ of the form $\mu f.\Lambda\overline{A}.\lambda x.\hat{t}$. Its decoding is a pair made of $\hat{v}$ and of a proof $\mathcal{H}$ asserting that $\hat{v}$ is a well-typed function closure. The encoding of that pair gives back the function closure. Reciprocally, let V be a value of type Func. This value must be a pair of a value $\hat{v}$ and of a proof $\mathcal{H}$ asserting that $\hat{v}$ is a well-typed function closure, so the encoding of V is the value $\hat{v}$, which is well-typed. The decoding of $\hat{v}$ gives back a pair made of $\hat{v}$ and of a proof $\mathcal{H}'$ asserting that $\hat{v}$ is a well-typed function closure. This pair is equal to V because the proof $\mathcal{H}'$ is equal to the proof $\mathcal{H}$. Indeed, by the proof-irrelevance property of the logic, two proofs of a same proposition are equal.

Second, consider a polymorphic value $\Lambda\overline{A}.\,\hat{v}$ of type $\forall\overline{A}.T$. The decoding of that value is the Coq value $\lambda\overline{A}.\,\lceil \hat{v} \rceil$ of type $\forall\overline{A}.\ \lceil\!\lceil T \rceil\!\rceil$. The encoding of that Coq value is written $\lfloor \lambda\overline{A}.\,\lceil \hat{v} \rceil \rfloor$. By definition of the encoding operator, it is equal to $\Lambda\overline{A}.\,\lfloor (\lambda\overline{A}.\,\lceil \hat{v} \rceil)\ \overline{A} \rfloor$. This value is equal to $\Lambda\overline{A}.\,\lfloor \lceil \hat{v} \rceil \rfloor$, and it therefore the same as the value $\Lambda\overline{A}.\,\hat{v}$, from which we started. Reciprocally, let V be a value of type $\forall\overline{A}.\mathcal{T}$.

[1] My earlier work on a deep embedding of Caml in Coq [15] involves a tool that, for every type constructor involved in a source Caml program, automatically generates the Coq definition of the encoding operator associated with that type.

The encoding of V is $\Lambda\overline{A}.\lfloor V\,\overline{A}\rfloor$. The decoding of this value, $\lceil\Lambda\overline{A}.\lfloor V\,\overline{A}\rfloor\rceil$ is equal to $\lambda\overline{A}.\lceil\lfloor V\,\overline{A}\rfloor\rceil$, which is the same as $\lambda\overline{A}.(V\,\overline{A})$. The latter is an eta-expansion of V, so it is equal to V, the value which we started from (recall that the target logic is assumed to feature functional extensionality). □

6.1.3 Substitution lemmas for weak-ML

Weak-ML does not enjoy type soundness, however the usual type-substitution and term-substitution lemmas hold. More precisely, typing derivations are preserved through instantiation of a type variable by a particular type, and they are preserved by substitution of a variable with a value of the appropriate type.

Lemma 6.1.2 (Type-substitution in weak-ML) *Let $\hat{t}$ be a typed term (or a typed value), let T be a type, and let Δ and Δ' be two typing contexts. Let $\overline{A}$ be a list of type variables, and let $\overline{T}$ be a list of types.*

$$\Delta, \overline{A}, \Delta' \vdash \hat{t}^T \quad \Rightarrow \quad \Delta, ([\overline{A} \to \overline{T}]\,\Delta') \vdash ([\overline{A} \to \overline{T}]\,\hat{t})^{([\overline{A}\to\overline{T}]\,T)}$$

Proof Straightforward by induction on the typing derivation. □

Lemma 6.1.3 (Term-substitution in weak-ML) *Let $\hat{t}$ be a typed term (or a typed value), let T be a type, let $\hat{w}$ be a polymorphic typed value of type S, and let Δ be a typing context.*

$$\Delta, x : S \vdash \hat{t}^T \quad \wedge \quad \Delta \vdash \hat{w}^S \quad \Rightarrow \quad \Delta \vdash ([x \to \hat{w}]\,\hat{t})^T$$

Proof By induction on the typing derivation. The interesting case occurs when $\hat{t}$ is the variable x applied to some types $\overline{T}$. Let $\Lambda\overline{A}.\hat{v}$ be the form of $\hat{w}$ and let $\forall\overline{A}.T'$ be the form of S. The hypothesis $\Delta \vdash \hat{w}^S$ implies $\Delta, \overline{A} \vdash \hat{v}^{T'}$. The assumption $\Delta, x : \forall\overline{A}.T' \vdash (x\,\overline{T})^T$ implies that T is equal to $[\overline{A} \to \overline{T}]\,T'$. The goal is to prove $\Delta \vdash ((\Lambda\overline{A}.\hat{v})\,\overline{T})^T$, which is the same as $\Delta \vdash ([\overline{A} \to \overline{T}]\,\hat{v})^{([\overline{A}\to\overline{T}]\,T')}$. This result follows from the type-substitution lemma applied to $\Delta, \overline{A} \vdash \hat{v}^{T'}$. □

6.1.4 Typed reductions

In this section, I introduce a reduction judgment for typed terms, written $\hat{t} \Downarrow \hat{v}$. Let me start by explaining why this judgment is needed.

Characteristic formulae are generated from typed values, and the strength of the hypotheses provided by characteristic formulae depend on the types, especially for functions. Consider the identity function "$\mathsf{let\,rec}\ f = \lambda x.\,x$". If this function is typed as a function from integers to integers, written "$\mathsf{let\,rec}\ f = \Lambda.\lambda x^{\mathsf{int}}.x$", then the body description of that function is:

$$\forall(X : \mathsf{int}).\,\forall(P : \mathsf{int} \to \mathsf{Prop}).\,P\,X \;\Rightarrow\; \mathsf{AppReturns}\,F\,X\,P$$

However, it the function is typed as a polymorphic function, written "$\mathsf{let\,rec}\, f = \Lambda A.\lambda x^A.x$", then the body description is a strictly stronger assertion:

$$\forall A.\,\forall (X : A).\,\forall (P : A \to \mathsf{Prop}).\,P\,X \;\Rightarrow\; \mathsf{AppReturns}\,F\,X\,P$$

This example suggests that the soundness and the completeness of characteristic formulae is strongly dependent on the types annotating the source code. The proofs of soundness and completeness relate a characteristic formula with a reduction judgment. In order to keep track of types in the proofs, I introduce a typed version of the reduction judgment, written $\hat{t} \Downarrow^{:} \hat{v}$. A derivation tree for a typed reduction $\hat{t} \Downarrow^{:} \hat{v}$ consists of a derivation tree describing not only the reduction steps involved but also the type of all the values involved throughout the execution.

The inductive rules, shown below, directly extend the rules defining the untyped reduction judgment "$t \Downarrow v$". When reducing a let-binding of the form "$\mathsf{let}\, x = \Lambda\overline{A}.\,\hat{t}_1\ \mathsf{in}\ \hat{t}_2$", the evaluation of $\hat{t}_1$ produces a typed value $\hat{v}_1$, and then the value x is replaced in $\hat{t}_2$ by the polymorphic value $\forall\overline{A}.\hat{v}_1$. When reducing a function definition "$\mathsf{let\,rec}\, f = \Lambda\overline{A}.\lambda x.\hat{t}_1\ \mathsf{in}\ \hat{t}_2$", the variable f is replaced in $\hat{t}_2$ with the function closure "$\mu f.\Lambda\overline{A}.\lambda x.\hat{t}_1$". Finally, consider a beta-redex of the form "$(\mu f.\Lambda\overline{A}.\lambda x.\hat{t}_1)\,\hat{v}_2$". This function reduces to the body $\hat{t}_1$ of the function in which the variable x is instantiated as $\hat{v}_2$, and the variable f is instantiated as a closure itself. Moreover, because applications are unconstrained in weak-ML, the typed reduction rule for beta-redexes includes hypotheses enforcing that the type of the argument $\hat{v}_2$ is an instance of the type T_0 of the argument x and that the type T of the entire redex is an instance of the type T_1 of the body of the function $\hat{t}_1$. The appropriate instantiation of the types $\overline{A}$ is uniquely determined by the other types involved, as established further on (Lemma 6.1.7). The reduction rules for conditionals are straightforward, so I do not show them.

Definition 6.1.1 (Typed reduction judgment)

$$\frac{\hat{t}_1 \Downarrow^{:} \hat{v}_1 \qquad ([x \to \Lambda\overline{A}.\,\hat{v}_1]\,\hat{t}_2) \Downarrow^{:} \hat{v}}{(\mathsf{let}\, x = \Lambda\overline{A}.\,\hat{t}_1\ \mathsf{in}\,\hat{t}_2) \Downarrow^{:} \hat{v}} \qquad \frac{([f \to \mu f.\Lambda\overline{A}.\lambda x.\hat{t}_1]\,\hat{t}_2) \Downarrow^{:} \hat{v}}{(\mathsf{let\,rec}\, f = \Lambda\overline{A}.\lambda x.\hat{t}_1\ \mathsf{in}\,\hat{t}_2) \Downarrow^{:} \hat{v}} \qquad \frac{}{\hat{v} \Downarrow^{:} \hat{v}}$$

$$\frac{T_2 = [\overline{A} \to \overline{T}]\,T_0 \quad T = [\overline{A} \to \overline{T}]\,T_1 \quad ([f \to \mu f.\Lambda\overline{A}.\lambda x.\hat{t}_1]\,[x \to \hat{v}_2]\,[\overline{A} \to \overline{T}]\,\hat{t}_1\,) \Downarrow^{:} \hat{v}}{((\mu f.\Lambda\overline{A}.\lambda x^{T_0}.\hat{t}_1^{\,T_1})^{\mathsf{func}}\,(\hat{v}_2^{\,T_2}))^{T} \Downarrow^{:} \hat{v}}$$

The next two lemmas relate the untyped reduction judgment for ML programs with the typed reduction judgment for weak-ML programs. A third lemma then establishes that the typed reduction of a well-typed term always produces a well-typed value of the same type, and a fourth lemma establishes that the typed reduction judgment is deterministic.

Lemma 6.1.4 (From typed reductions to untyped reductions) *Let $\hat{t}$ be a typed term and $\hat{v}$ be a typed value, both carrying weak-ML type annotations. Let t*

and v be the terms obtained by stripping types out of $\hat{t}$ and $\hat{v}$, respectively.

$$\hat{t} \Downarrow^{:} \hat{v} \qquad \Rightarrow \qquad t \Downarrow v$$

Proof Removing the type annotation and the type substitutions from the rules defining the typed reduction judgment give exactly the rules defining the untyped reduction judgment. □

Lemma 6.1.5 (From untyped reductions to typed reductions, in ML) *Consider a well-typed ML term, fully annotated with ML types. Let $\hat{t}$ be the corresponding term where ML type annotations are turned into weak-ML type annotations, by application of the operator $\langle\cdot\rangle$, and let t be the corresponding term in which all type annotations are removed. Assume there exists a value v such that t reduces to v, that is, such that $t \Downarrow v$. Then, there exists a typed value $\hat{v}$, annotated with weak-ML types, that corresponds to the value v and such that $\hat{t} \Downarrow^{:} \hat{v}$.*

Proof Let $\mathring{t}$ denote a term annotated with ML types. Following the subject reduction proof for ML, one can show that the untyped reduction sequence $t \Downarrow v$ can be turned into a typed reduction sequence $\mathring{t} \Downarrow \mathring{v}$, which is a judgment defined like $\hat{t} \Downarrow^{:} \hat{v}$ except that it involves terms and values are annotated with ML types instead of weak-ML types. Then, applying the operator $\langle\cdot\rangle$ to the entire derivation $\mathring{t} \Downarrow^{:} \mathring{v}$ produces exactly the required derivation $\hat{t} \Downarrow^{:} \hat{v}$. □

Lemma 6.1.6 (Typed reductions preserve well-typedness) *Let $\hat{t}$ be a typed term, let $\hat{v}$ be a typed value, let T be a type and let $\overline{B}$ denote a set of free type variables.*

$$\overline{B} \vdash \hat{t}^{T} \quad \wedge \quad \hat{t} \Downarrow^{:} \hat{v} \quad \Rightarrow \quad \overline{B} \vdash \hat{v}^{T}$$

Proof By induction on the typed-reduction derivation. I only show the proof case for let-bindings, to illustrate the kind of arguments involved, as well as the proof case for the beta-reduction rule, which is the only one specific to weak-ML.

• Case $\hat{t}$ is of the form $(\mathsf{let}\, x = \Lambda\overline{A}.\, \hat{t}_1{}^{T_1}\, \mathsf{in}\, \hat{t}_2{}^{T_2})^{T_2}$ and reduces to $\hat{v}$. The premises assert that $\hat{t}_1 \Downarrow^{:} \hat{v}_1$ and $([x \to \Lambda\overline{A}.\, \hat{v}_1]\, \hat{t}_2) \Downarrow^{:} \hat{v}$ hold. By hypothesis, $\hat{t}_1{}^{T_1}$ is well-typed in the context $(\overline{B}, \overline{A})$ and $\hat{t}_2{}^{T_2}$ is well-typed in the context $(\overline{B}, x : \forall\overline{A}.T_1)$. By induction hypothesis, $\hat{v}_1{}^{T_1}$ is also well-typed in that context So, $\Lambda\overline{A}.\, \hat{v}_1$ admits the type $\forall\overline{A}.T_1$ in the context $\overline{B}$. By the substitution lemma applied to the typing assumption for $\hat{t}_2{}^{T_2}$, the term $([x \to \Lambda\overline{A}.\, \hat{v}_1]\, \hat{t}_2{}^{T_2})$ also admits the type T_2 in the context $\overline{B}$. Therefore, by induction hypothesis, $\hat{v}$ admits the type T_2 in the context $\overline{B}$.

• Case $\hat{t}$ is of the form $(\mu f.\Lambda\overline{A}.\lambda x^{T_0}.\hat{t}_1{}^{T_1}\, (\hat{v}_2{}^{T_2}))^{T}$ and reduces to $\hat{v}$. The premises assert that the propositions $T_2 = [\overline{A} \to \overline{T}]\, T_0$ and $T = [\overline{A} \to \overline{T}]\, T_1$ hold and that $[f \to \mu f.\Lambda\overline{A}.\lambda x.\hat{t}_1]\, [x \to \hat{v}_2]\, [\overline{A} \to \overline{T}]\, \hat{t}_1$ reduces to $\hat{v}$. The goal is to prove that the typed term $([f \to \mu f.\Lambda\overline{A}.\lambda x.\hat{t}_1]\, [x \to \hat{v}_2]\, [\overline{A} \to \overline{T}]\, \hat{t}_1)^{T}$ is well-typed in the context $\overline{B}$. By inversion on the typing rule for applications, we obtain the fact that both $\hat{v}_2{}^{T_2}$ and $(\mu f.\Lambda\overline{A}.\lambda x^{T_0}.\hat{t}_1{}^{T_1})^{\mathsf{func}}$ are well-typed. By inversion on the typing rule for

function closures, we obtain the typing assumption "$\overline{B}, \overline{A}, f : \mathsf{func}, x : T_0 \vdash \hat{t}_1^{T_1}$". By the type substitution lemma, this implies "$\overline{B}, f : \mathsf{func}, x : ([\overline{A} \to \overline{T}]\, T_0) \vdash ([\overline{A} \to \overline{T}]\, \hat{t}_1)^{([\overline{A} \to \overline{T}]\, T_1)}$". Exploiting the two assumptions $T_2 = [\overline{A} \to \overline{T}]\, T_0$ and $T = [\overline{A} \to \overline{T}]\, T_1$, we can rewrite this proposition as "$\overline{B}, f : \mathsf{func}, x : T_2 \vdash ([\overline{A} \to \overline{T}]\, \hat{t}_1)^T$". The conclusion, which is "$\overline{B} \vdash ([f \to \mu f.\Lambda\overline{A}.\lambda x.\hat{t}_1]\, [x \to \hat{v}_2]\, [\overline{A} \to \overline{T}]\, \hat{t}_1)^T$", can then be deduced by applying the substitution lemma twice, once for x and once for f. □

Lemma 6.1.7 (Determinacy of typed reductions) *Let $\hat{t}$ be a typed term, and let $\hat{v}_1$ and $\hat{v}_2$ be two typed values.*

$$\hat{t} \Downarrow^{:} \hat{v}_1 \quad \wedge \quad \hat{t} \Downarrow^{:} \hat{v}_2 \quad \Rightarrow \quad \hat{v}_1 = \hat{v}_2$$

Proof By induction on the first hypothesis and case analysis on the second one. The nontrivial case is that of applications, for which we need to establish that the list of types $\overline{T}$ is uniquely determined. Consider a list $\overline{T}$ such that $T_2 = [\overline{A} \to \overline{T}]\, T_0$ and $T = [\overline{A} \to \overline{T}]\, T_1$, and another list $\overline{T'}$ such that $T_2 = [\overline{A} \to \overline{T'}]\, T_0$ and $T = [\overline{A} \to \overline{T'}]\, T_1$. Then, $[\overline{A} \to \overline{T}]\, T_0 = [\overline{A} \to \overline{T'}]\, T_0$ and $[\overline{A} \to \overline{T}]\, T_1 = [\overline{A} \to \overline{T'}]\, T_1$. Since the free variables $\overline{A}$ are included in the union of the set of free type variables of T_0 and of that of T_1, the list $\overline{T'}$ must be equal to the list $\overline{T}$. □

6.1.5 Interpretation and properties of AppEval

Recall that AppEval is the low-level predicate in terms of which AppReturns is defined. In this section, I give an interpretation of AppEval in terms of the typed semantics of the source language. The type of AppEval is recalled next.

$$\mathsf{AppEval} \;:\; \forall AB.\, \mathsf{Func} \to A \to B \to \mathsf{Prop}$$

The proposition $\mathsf{AppEval}\, F\, V\, V'$ asserts that the application of a Caml function reflected as F in the logic to a value reflected as V in the logic terminates and returns a Caml value reflected as V' in the logic. This can be reformulated using encoders, saying that the application of the encoding of F to the encoding of V terminates and returns the encoding of V'. Hence the following definition of the predicate AppEval.

Definition 6.1.2 (Definition of AppEval) *Let F be a value of type Func, V is a value of type $\mathcal{T}$ and V' is a value of type $\mathcal{T}'$.*

$$\mathit{AppEval}\, F\, V\, V' \quad \equiv \quad (\lfloor F \rfloor\, \lfloor V \rfloor) \Downarrow^{:} \lfloor V' \rfloor$$

Remark: one can also present this definition as an inductive rule, as follows.

$$\frac{(\hat{f}\, \hat{v}) \Downarrow^{:} \hat{v}'}{\mathsf{AppEval}\, \lceil \hat{f} \rceil\, \lceil \hat{v} \rceil\, \lceil \hat{v}' \rceil}$$

The semantics of the source language is assumed to be deterministic. Since AppEval lifts the semantics of function application to the level of Coq values, for every function F and argument V there is at most one result value V' for which the relation $\mathsf{AppEval}\, F\, V\, V'$ holds. This property is formally captured through the following lemma.

Lemma 6.1.8 (Determinacy of AppEval) *For any types $\mathcal{T}$ and $\mathcal{T}'$, for any Coq value F of type Func, any Coq value V of type $\mathcal{T}$, and any two Coq values V_1' and V_2' of type $\mathcal{T}'$,*

$$\mathsf{AppEval}\,F\,V\,V_1' \quad \wedge \quad \mathsf{AppEval}\,F\,V\,V_2' \quad \Rightarrow \quad V_1' = V_2'$$

Proof By definition of AppEval, the hypotheses are equivalent to $(\lfloor F \rfloor\, \lfloor V \rfloor) \Downarrow^{:} \lfloor V_1' \rfloor$ and $(\lfloor F \rfloor\, \lfloor V \rfloor) \Downarrow^{:} \lfloor V_2' \rfloor$. By determinacy of typed reductions (Lemma 6.1.7), $\lfloor V_1' \rfloor$ is equal to $\lfloor V_2' \rfloor$. The equality between V_1' and V_2' then follows from the injectivity of encoders. □

Remark: ideally, the soundness theorem for characteristic formulae should be proved without exploiting the determinacy assumption on the source language. However, the proof appears to become slightly more complicated when this assumption is not available. For this reason, I leave the generalization of the soundness proof to a non-deterministic language for future work.

6.1.6 Substitution lemmas for characteristic formulae

The substitution lemma for characteristic formulae is used both in the proof of soundness and in the proof of completeness. It explains how a substitution of a value for a variable in a term commutes with the computation of the characteristic formula for that term. The proof of the substitution lemma involves a corresponding substitution lemma for the decoding operator. It also relies on two type-substitution lemmas, one for characteristic formulae and one for the decoding operator, presented next.

Lemma 6.1.9 (Type-substitution lemmas) *Let $\hat{v}$ be a well-typed value and $\hat{t}$ be a well-typed term in a context $(\Delta, \overline{A})$. Let Γ be a decoding context with types corresponding to those in Δ. Let $\overline{T}$ be a list of weak-ML types of the same arity as $\overline{A}$.*

$$\begin{aligned} \lceil [\overline{A} \to \overline{T}]\, \hat{v} \rceil^{\Gamma} &= [A \to \llbracket \overline{T} \rrbracket\ ]\, \lceil \hat{v} \rceil^{\Gamma} \\ \llbracket\, [\overline{A} \to \overline{T}]\, \hat{t}\, \rrbracket^{\Gamma} &= [A \to \llbracket \overline{T} \rrbracket\ ]\, \llbracket\, \hat{t}\, \rrbracket^{\Gamma} \end{aligned}$$

Proof By induction on the structure of $\hat{v}$ and $\hat{t}$. □

A useful corollary to the type-substitution lemma for decoders describes the application of the decoding of a polymorphic value. Recall that $\hat{w}$ ranges over polymorphic values. I use the notation $\hat{w}\,\overline{T}$ to denote the type application of $\hat{w}$ to the types $\overline{T}$: if $\hat{w}$ is of the form $\Lambda\overline{A}.\,\hat{v}$, then $\hat{w}\,\overline{T}$ is equal to $[\overline{A} \to \overline{T}]\,\hat{v}$. Using this notation, the commutation of decoding with type applications of polymorphic values is formalized as follows.

Lemma 6.1.10 (Application of the decoding of polymorphic value) *Let $\hat{w}$ be a closed polymorphic value of type $\forall\overline{A}.T$, and let $\overline{T}$ be a list of types of the same arity as $\overline{A}$.*

$$\lceil \hat{w}\,\overline{T} \rceil \quad = \quad \lceil \hat{w} \rceil\ \llbracket \overline{T} \rrbracket$$

Proof Let $\Lambda\overline{A}.\,\hat{v}$ be the form of $\hat{w}$. The left-hand side is equal to $\lceil[\overline{A} \to \overline{T}]\,\hat{v}\rceil$. The right-hand side is equal to "$(\lambda\overline{A}.\,\lceil\hat{v}\rceil)\ [\![\overline{T}]\!]$", which is the same as $[\overline{A} \to [\![\overline{T}]\!]]\,\lceil\hat{v}\rceil$. The conclusion then follows from the type-substitution lemma (Lemma 6.1.9). □

Lemma 6.1.11 (Substitution lemma for decoding) *Let $\hat{v}$ be a value of type T with a free variable x of type S, let $\hat{w}$ be a closed value of type S, and let Γ be a decoding context. Then,*

$$\lceil\,[x \to \hat{w}]\,\hat{v}\,\rceil^{\Gamma} \;=\; \lceil\,\hat{v}\,\rceil^{(\Gamma, x \mapsto \lceil\hat{w}\rceil)}$$

Proof By induction on the structure of $\hat{v}$. The interesting case is when $\hat{v}$ is an occurrence of the variable x. An occurrence of x in $\hat{v}$ take the form of a type application $x\,\overline{T}$. On the left-hand side, the substitution of x by $\hat{w}$ gives $\hat{w}\,\overline{T}$. This value is then decoded as $\lceil\hat{w}\,\overline{T}\rceil$. On the right-hand side, $x\,\overline{T}$ is directly decoded as $\lceil\hat{w}\rceil\ [\![\overline{T}]\!]$, since x is mapped to $\lceil\hat{w}\rceil$ is the decoding context $(\Gamma, x \mapsto \lceil\hat{w}\rceil)$. Using the previous lemma, we can conclude that both sides yield equal values. □.

Lemma 6.1.12 (Substitution lemma for characteristic formulae) *Let $\hat{t}$ be a well-typed term with a free variable x of type S, and let $\hat{w}$ be a closed typed value of type S. Then,*

$$[\![\,[x \to \hat{w}]\,\hat{t}\,]\!]^{\Gamma} \;=\; [\![\,\hat{t}\,]\!]^{(\Gamma, x \mapsto \lceil\hat{w}\rceil)}$$

Proof By induction on the structure of $\hat{t}$, invoking the previous lemma when reaching values. □

6.1.7 Weakening lemma

The following weakening lemma is involved in the proof of completeness.

Lemma 6.1.13 (Weakening for characteristic formulae)
For any well-typed term $\hat{t}$, for any post-conditions P and P', and for any context Γ,

$$(\forall X.\, P\,X \;\Rightarrow\; P'\,X) \quad\wedge\quad [\![\,\hat{t}\,]\!]^{\Gamma}\,P \quad\Rightarrow\quad [\![\,\hat{t}\,]\!]^{\Gamma}\,P'$$

Proof By induction on $\hat{t}$. The only nontrivial case is that for applications, where, given a function F and an argument V, we must show that $\mathsf{AppReturns}FVP$ implies $\mathsf{AppReturns}FVP'$. By definition of $\mathsf{AppReturns}$, the assumption ensures the existence of a value V' such that $\mathsf{AppEval}\,F\,V\,V'$ and $P\,V'$. Therefore, the same value V' is such that $\mathsf{AppEval}\,F\,V\,V'$ and $P'\,V'$. Thus the proposition $\mathsf{AppReturns}FVP'$ holds. □

6.1.8 Elimination of n-ary functions

For the sake of verifying programs in practice, I have introduced a direct treatment of n-ary functions, with the predicate Spec. However, for the sake of conducting proofs, it is much simpler to consider only unary functions, with characteristic formulae

expressed directly in terms of $\mathsf{AppReturns}$. In this section, I formally justify that the equivalence between the two presentations.

For functions of arity one, I show the equivalence between the body description of a function "$\mathsf{let\,rec}\, f = \lambda x.\, t$" stated in terms of Spec_1 and the body description of that same function stated in terms of $\mathsf{AppReturns}_1$. In the corresponding lemma, shown below, $(G\,x)$ denotes the characteristic formula of the body t of the function "$\mathsf{let\,rec}\, f = \lambda x.\, t$" (the characteristic formula of t indeed depends on the variable x).

Lemma 6.1.14 (Spec_1-to-$\mathsf{AppReturns}_1$)

$$\begin{array}{ll} & (\forall K.\ \mathit{is\_spec}_1\, K \wedge (\forall x.\ K\,x\,(G\,x)) \Rightarrow \mathit{Spec}_1 f K) \\ \iff & (\forall x\,P.\ (G\,x)\,P \Rightarrow \mathit{AppReturns}_1\, f\,x\,P) \end{array}$$

Proof The proof is not entirely straightforward, so I have also carried it out in Coq. To prove the proposition "$\mathsf{AppReturns}_1\, f\, v\, P$" under the assumption "$(G\,v)\,P$", we instantiate K as "$\lambda x\,R.\, x = v \Rightarrow R\,P$" and we are left proving $\mathsf{is\_spec}_1\, K$, which holds because K is covariant in R, as well as "$\forall x.\ K\,x\,(G\,x)$", which is equivalent to "$x = v \Rightarrow (G\,x)\,P$". The latter directly follows from the assumption "$(G\,v)\,P$". Reciprocally, we need to prove "$\mathsf{Spec}_1 f K$" under the assumptions "$\mathsf{is\_spec}_1\, K$" and "$\forall x.\ K\,x\,(G\,x)$" and "$\forall x\,P.\ (G\,x)\,P \Rightarrow \mathsf{AppReturns}_1\, f\,x\,P$". By definition of Spec_1, and given that $\mathsf{is\_spec}_1\, K$ is true by assumption, it suffices to prove "$\forall x.\ K\,x\,(\mathsf{AppReturns}_1\, f\,x)$". The hypothesis "$\mathsf{is\_spec}_1\, K$" asserts that K is covariant in its second argument. So, exploiting the assumption "$\forall x.\ K\,x\,(G\,x)$", it remains to prove "$\forall x\,P.\ (G\,x)\,P \Rightarrow \mathsf{AppReturns}_1\, f\,x\,P$", which corresponds exactly to one of the assumptions. □

For a function of arity two, I show the equivalence between the body description of a function "$\mathsf{let\,rec}\, f = \lambda x\,y.\, t$" and the body description of the function "$\mathsf{let\,rec}\, f = \lambda x.\, (\mathsf{let\,rec}\, g = \lambda y.\, t\,\mathsf{in}\, y)$". This encoding of n-ary functions as unary function can be easily extended to higher arities. In the corresponding lemma, shown below, $(G\,x\,y)$ denotes the characteristic formula of the body t.

Lemma 6.1.15 (Spec_2-to-$\mathsf{AppReturns}_1$)

$$\begin{array}{ll} & (\forall K.\ \mathit{is\_spec}_2\, K \wedge (\forall x\,y.\ K\,x\,y\,(G\,x\,y)) \Rightarrow \mathit{Spec}_2 f K) \\ \iff & (\forall x\,P.\ (\forall g.\ \mathcal{H} \Rightarrow P\,g) \Rightarrow \mathit{AppReturns}_1\, f\,x\,P) \\ & \textit{where}\ \mathcal{H} \equiv (\forall y\,P'.\ (G\,x\,y)\,P' \Rightarrow \mathit{AppReturns}_1\, g\,y\,P') \end{array}$$

The proof of this lemma is even more technical than the previous one because it involves exploiting the determinacy of partial applications. So, I do not show the details and refer the reader to the Coq proof for further details. Note that the treatment of arities higher than two does not involve further difficulties. It simply involves additional proof steps for reasoning on iterated partial applications.

6.2 Soundness

6.2.1 Soundness of characteristic formulae

The soundness theorem states that if the characteristic formula of a well-typed term $\hat{t}$ holds of a post-condition P, then there exists a value $\hat{v}$ such that $\hat{t}$ evaluates to $\hat{v}$ and such that the decoding of $\hat{v}$ satisfies P. Note that representation predicates are defined in Coq and their properties are proved in Coq, so they are not involved at any point in the soundness proof.

Theorem 6.2.1 (Soundness) *Let $\hat{t}$ be a closed typed term, let T be the type of $\hat{t}$, and let P be a predicate of type $[\![T]\!] \to \mathsf{Prop}$.*

$$\vdash \hat{t} \quad \wedge \quad [\![\hat{t}]\!]^{\emptyset}\, P \qquad \Rightarrow \qquad \exists \hat{v}.\quad \hat{t} \Downarrow^{:} \hat{v} \quad \wedge \quad P\, \lceil \hat{v} \rceil$$

Proof The proof goes by induction on the size of $\hat{t}$. The size of a term is defined as the number of nodes from the grammar of terms. Any value contained in a term is considered to have size one. In particular, a function closure has size one, even though its body is a term whose size may be greater than one. Note that this non-standard definition of the size of the term plays a crucial role in the proof. Indeed, the standard definition of the size of the term would not enable the induction principle to be applied because β-reduction may cause terms to grow in size.

- Case $\hat{t} = \hat{v}$. The assumption $[\![\hat{t}]\!]^{\emptyset}\, P$ is equivalent to "$P\, \lceil \hat{v} \rceil$". The conclusion follows immediately, since $\hat{v} \Downarrow^{:} \hat{v}$.
- Case $\hat{t} = (((\hat{f}^{\mathsf{func}})\, (\hat{v}'^{T'}))^{T})$. The assumption coming from the characteristic formula is "$\mathsf{AppReturns}\, \lceil \hat{f} \rceil\, \lceil \hat{v}' \rceil\, P$". The goal is to find a value $\hat{v}$ such that $(\hat{f}\, \hat{v}') \Downarrow^{:} \hat{v}$ and "$P\, \lceil \hat{v} \rceil$". The assumption asserts the existence of a value V of type $[\![T]\!]$ such that "$P\, V$" holds and such that "$\mathsf{AppEval}\, \lceil \hat{f} \rceil\, \lceil \hat{v}' \rceil\, V$" holds. By definition of the predicate $\mathsf{AppEval}$, we have $(\hat{f}\, \hat{v}') \Downarrow^{:} \lfloor V \rfloor$. To conclude, I instantiate $\hat{v}$ as $\lfloor V \rfloor$. So, V is equal to $\lceil \hat{v} \rceil$, the proposition "$P\, \lceil \hat{v} \rceil$" follows from "$P\, V$".
- Case $\hat{t} = \mathsf{crash}$. The assumption is False, so there is nothing to prove.
- Case $\hat{t} = \mathsf{if}\, \lceil \hat{v} \rceil\, \mathsf{then}\, \hat{t}_1\, \mathsf{else}\, \hat{t}_2$. The assumption is as follows.

$$(\lceil \hat{v} \rceil = \mathsf{true} \Rightarrow [\![\hat{t}_1]\!]\, P) \quad \wedge \quad (\lceil \hat{v} \rceil = \mathsf{false} \Rightarrow [\![\hat{t}_2]\!]\, P)$$

The value $\hat{v}$ is of type bool, so it is either equal to true or to false. Assume that $\hat{v}$ is equal to true. The first part of the assumption gives $[\![\hat{t}_1]\!]\, P$. The conclusions then directly follow from the induction hypothesis applied to that fact. The case where $\hat{v}$ is equal to false is symmetrical.

- Case $\hat{t} = (\mathsf{let\,rec}\, f = \Lambda \overline{A}.\lambda x^{T_0}.\hat{t}_1{}^{T_1}\ \mathsf{in}\ \hat{t}_2{}^{T_2})$. I start with the case where the function is not polymorphic, that is, when $\overline{A}$ is empty. The generalization to polymorphic functions is given afterwards. The assumption is the proposition $\forall F.\, \mathcal{H} \Rightarrow [\![\hat{t}_2]\!]^{(f \mapsto F)}\, P$, where $\mathcal{H}$ stands for:

$$\forall X.\, \forall P'.\, [\![\hat{t}_1]\!]^{(x \mapsto X, f \mapsto F)}\, P' \Rightarrow \mathsf{AppReturns}\, F\, X\, P'$$

I instantiate the assumption with F as $\lceil \mu f.\lambda x.\hat{t}_1 \rceil$. According to the assumption, $\mathcal{H}$ implies $[\![\hat{t}_2]\!]^{(f \mapsto F)}\, P$. The plan is to prove that $\mathcal{H}$ holds, so as to later exploit the hypothesis $[\![\hat{t}_2]\!]^{(f \mapsto F)}\, P$.

To prove $\mathcal{H}$, consider some arbitrary arguments X and P'. The hypothesis is $([\![\hat{t}_1]\!]^{(x \mapsto X, f \mapsto F)}\ P')$. The goal consists in proving $\mathsf{AppReturns}\, F\, X\, P'$. Since X has type $[\![T_0]\!]$, we can consider its encoding. Let $\hat{v}_2$ denote the value $\lfloor X \rfloor$, which has type T_0. We know that X is equal to $\lceil \hat{v}_2 \rceil$ and that X has type $[\![T_0]\!]$. So, the assumption becomes $([\![\hat{t}_1]\!]^{(x \mapsto \lceil \hat{v}_2 \rceil, f \mapsto \lceil \mu f.\lambda x.\hat{t}_1 \rceil)}\ P')$, which, by the substitution lemma, gives $([\![[f \to \mu f.\lambda x.\hat{t}_1]\,[x \to \hat{v}_2]\,\hat{t}_1]\!]^{\emptyset}\, P')$.

By induction hypothesis applied to the term "$[f \to \mu f.\lambda x.\hat{t}_1]\,[x \to \hat{v}_2]\,\hat{t}_1$", of type T_1, there exists a value $\hat{v}'$ such that the term considered reduces to $\hat{v}'$ and such that "$P'\,\lceil \hat{v}' \rceil$" holds. We can deduce the typed reduction judgment "$((\mu f.\lambda x.\hat{t}_1)\,\hat{v}_2) \Downarrow^{:} \hat{v}'$". Exploiting the definition of $\mathsf{AppEval}$, we deduce $\mathsf{AppEval}\, \lceil \mu f.\lambda x.\hat{t}_1 \rceil\, \lceil \hat{v}_2 \rceil\, \lceil \hat{v}' \rceil$. The conclusion of $\mathcal{H}$, namely $\mathsf{AppReturns}\, F\, X\, P'$, follows from the latter proposition and from the assumption "$P'\,\lceil \hat{v}' \rceil$".

Finally, it remains to exploit the assumption $[\![\hat{t}_2]\!]^{(f \mapsto F)}\, P$ in order to conclude. By the substitution lemma, this fact can be reformulated as $[\![\,[f \to \mu f.\lambda x.\hat{t}_1]\,\hat{t}_2\,]\!]^{\emptyset}\, P$. The conclusion follows from the induction hypothesis applied to that fact.

• Case "$\hat{t} = (\mathsf{let}\, x = \Lambda \overline{A}.\, \hat{t}_1\ \mathsf{in}\ \hat{t}_2)$". Let T_1 denote the type of $\hat{t}_1$. I start by giving the proof in the particular case where x has a monomorphic type, that is, when $\overline{A}$ is empty. The characteristic formula asserts that there exists a predicate P' of type "$[\![T_1]\!] \to \mathsf{Prop}$" such that "$[\![\hat{t}_1]\!]^{\emptyset}\, P'$" and such that, for any X of type $[\![T_1]\!]$ satisfying the predicate P', the proposition "$[\![\hat{t}_2]\!]^{(x \mapsto X)}\ P$" holds. The goal is to find a value $\hat{v}_1$ and a value $\hat{v}$ such that $\hat{t}_1$ reduces to $\hat{v}_1$ and "$[x \to \hat{v}_1]\,\hat{t}_2$" reduces to $\hat{v}$, and such that "$P\,\lceil \hat{v} \rceil$" holds.

By induction hypothesis applied to "$[\![\hat{t}_1]\!]^{\emptyset}\, P'$", there exists a value $\hat{v}_1$ such that $\hat{t}_1 \Downarrow^{:} \hat{v}_1$ and $P'\,\lceil \hat{v}_1 \rceil$. Because typed reduction preserves typing, the value $\hat{v}_1$ has the same type as $\hat{t}_1$, namely T_1. Instantiating the variable X from the assumption with the value $\lceil \hat{v}_1 \rceil$ gives "$[\![\hat{t}_2]\!]^{(x \mapsto \lceil \hat{v}_1 \rceil)}\, P$". By the substitution lemma, this proposition is equivalent to "$[\![[x \to \hat{v}_1]\,\hat{t}_2]\!]^{\emptyset}\, P$". Note that the term $[x \to \hat{v}_1]\,\hat{t}_2$ is well-typed by the substitution lemma. The application of the induction hypothesis to the term $[x \to \hat{v}_1]\,\hat{t}_2$ asserts the existence of a value $\hat{v}$ of type T such that $([x \to \hat{v}_1]\,\hat{t}_2) \Downarrow^{:} \hat{v}$ and such that $P\,\lceil \hat{v} \rceil$, as required.

Generalization to polymorphic let-bindings Let S be a shorthand for $\forall \overline{A}.T_1$, which is the type of x. The characteristic formula asserts that there exists a predicate P' of type "$\forall \overline{A}.([\![T_1]\!] \to \mathsf{Prop})$" such that "$\forall \overline{A}.\ [\![\hat{t}_1]\!]^{\emptyset}\ (P'\,\overline{A})$" holds, and such that, for any X of type $[\![S]\!]$ satisfying the proposition "$\forall \overline{A}.\ (P'\,\overline{A}\,(X\,\overline{A}))$", the proposition "$[\![\hat{t}_2]\!]^{(x \mapsto X)}\ P$" holds. The goal is to find a value $\hat{v}_1$ and a value $\hat{v}$ such that $\hat{t}_1$ reduces to $\hat{v}_1$ and "$[x \to \Lambda \overline{A}.\, \hat{v}_1]\,\hat{t}_2$" reduces to $\hat{v}$, and such that "$P\,\lceil \hat{v} \rceil$" holds.

Let $\overline{A}$ be some type variables. The induction hypothesis for "$[\![\hat{t}_1]\!]^{\emptyset}\ (P'\,\overline{A})$" asserts there exists a value $\hat{v}_1$ of type T_1 such that $\hat{t}_1 \Downarrow^{:} \hat{v}_1$ and $(P'\,\overline{A})\,\lceil \hat{v}_1 \rceil$. Now, I instantiate the variable X from the assumption with the polymorphic value $\lceil \Lambda \overline{A}.\, \hat{v}_1 \rceil$, of type

$\forall \overline{A}.\llbracket T_1 \rrbracket$. Note that, by definition of the decoding of polymorphic values, X is also equal to $\lambda \overline{A}.\, \lceil \hat{v}_1 \rceil$.

At this point, we need to prove the premise "$\forall \overline{A}.\ (P'\,\overline{A})\,(X\,\overline{A})$". Let $\overline{\mathcal{T}}$ be some type variables. The goal is to prove "$(P'\,\overline{\mathcal{T}})\,(X\,\overline{\mathcal{T}})$". Substituting $\overline{A}$ with $\overline{\mathcal{T}}$ in the proposition $(P'\,\overline{A})\,\lceil \hat{v}_1 \rceil$ gives $(P'\,\overline{\mathcal{T}})\,([\overline{A} \to \overline{\mathcal{T}}]\,\lceil \hat{v}_1 \rceil)$. To conclude, it suffices to prove that $X\,\overline{\mathcal{T}}$ is equal to $[\overline{A} \to \overline{\mathcal{T}}]\,\lceil \hat{v}_1 \rceil$. This fact holds because X is equal to $\lambda \overline{A}.\, \lceil \hat{v}_1 \rceil$.

Now that we have proved the premise "$\forall \overline{A}.\ (P'\,\overline{A})\,(X\,\overline{A})$", we need to find the value $\hat{v}$ to which "$[x \to \Lambda \overline{A}.\, \hat{v}_1]\,\hat{t}_2$" reduces. Given the instantiation of X, the assumption "$\llbracket \hat{t}_2 \rrbracket^{(x \mapsto X)}\ P$" is equivalent to "$\llbracket \hat{t}_2 \rrbracket^{(x \mapsto \lceil \Lambda \overline{A}.\, \hat{v}_1 \rceil)}\ P$". By the substitution lemma, this proposition is equivalent to "$\llbracket [x \to \Lambda \overline{A}.\, \hat{v}_1]\,\hat{t}_2 \rrbracket^{\emptyset}\ P$". The conclusion then follows from the application of the induction hypothesis to the term $[x \to \Lambda \overline{A}.\, \hat{v}_1]\,\hat{t}_2$.

Generalization to polymorphic functions This proof directly generalizes that given for non-polymorphic functions. As before, the term $\hat{t}$ is of the form $(\mathsf{let\ rec}\ f = \Lambda \overline{A}.\lambda x^{T_0}.\hat{t}_1{}^{T_1}\ \mathsf{in}\ \hat{t}_2{}^{T_2})$. The assumption is the proposition $\forall F.\ \mathcal{H} \Rightarrow \llbracket \hat{t}_2 \rrbracket^{(f \mapsto F)}\, P$, where $\mathcal{H}$ stands for:

$$\forall \overline{A}.\, \forall X.\, \forall P'.\, \llbracket \hat{t}_1 \rrbracket^{(x \mapsto X, f \mapsto F)}\ P' \Rightarrow \mathsf{AppReturns}\, F\, X\, P'$$

I instantiate the assumption with F as $\lceil \mu f.\Lambda \overline{A}.\lambda x.\hat{t}_1 \rceil$. According to the assumption, $\mathcal{H}$ implies $\llbracket \hat{t}_2 \rrbracket^{(f \mapsto F)}\, P$. The plan is to prove that $\mathcal{H}$ holds, so as to later exploit the hypothesis $\llbracket \hat{t}_2 \rrbracket^{(f \mapsto F)}\, P$.

To prove $\mathcal{H}$, consider some arbitrary arguments $\overline{\mathcal{T}}$, X and P'. Let $\overline{T}$ be a shorthand for $\lfloor \overline{\mathcal{T}} \rfloor$. The hypothesis is $(\llbracket [\overline{A} \to \overline{T}]\,\hat{t}_1 \rrbracket^{(x \mapsto X, f \mapsto F)}\ P')$. The goal consists in proving $\mathsf{AppReturns}\, F\, X\, P'$, where X has type $\llbracket [\overline{A} \to \overline{T}]\, T_0 \rrbracket$ and P' has type $\llbracket [\overline{A} \to \overline{T}]\, T_1 \rrbracket \to \mathsf{Prop}$. Let $\hat{v}_2$ denote the value $\lfloor X \rfloor$. Since X has type $\llbracket [\overline{A} \to \overline{T}]\, T_0 \rrbracket$, the value $\hat{v}_2$ has type $[\overline{A} \to \overline{T}]\, T_0$. We know that X is equal to $\lceil \hat{v}_2 \rceil$ and that X has type $\llbracket T_0 \rrbracket$. The assumption becomes $(\llbracket [\overline{A} \to \overline{T}]\,\hat{t}_1 \rrbracket^{(x \mapsto \lceil \hat{v}_2 \rceil, f \mapsto \lceil \mu f.\Lambda \overline{A}.\lambda x.\hat{t}_1 \rceil)}\ P')$, which, by the substitution lemma, gives $(\llbracket [f \to \mu f.\Lambda \overline{A}.\lambda x.\hat{t}_1]\,[x \to \hat{v}_2]\,[\overline{A} \to \overline{T}]\,\hat{t}_1 \rrbracket^{\emptyset}\, P')$.

By induction hypothesis applied to the term "$[f \to \mu f.\Lambda \overline{A}.\lambda x.\hat{t}_1]\,[x \to \hat{v}_2]\,[\overline{A} \to \overline{T}]\,\hat{t}_1$", of type $[\overline{A} \to \overline{T}]\, T_1$, there exists a value $\hat{v}'$ such that the term considered reduces to $\hat{v}'$ and such that "$P'\,\lceil \hat{v}' \rceil$" holds. We can deduce the typed reduction judgment "$((\mu f.\Lambda \overline{A}.\lambda x.\hat{t}_1)\,\hat{v}_2) \Downarrow^{:} \hat{v}'$", because the argument $\hat{v}_2$ has the type $[\overline{A} \to \overline{T}]\, T_0$ and the result $\hat{v}'$ is of type $[\overline{A} \to \overline{T}]\, T_1$. Exploiting the definition of $\mathsf{AppEval}$, we can deduce $\mathsf{AppEval}\,\lceil \mu f.\Lambda \overline{A}.\lambda x.\hat{t}_1 \rceil\,\lceil \hat{v}_2 \rceil\,\lceil \hat{v}' \rceil$. The conclusion of $\mathcal{H}$, namely $\mathsf{AppReturns}\, F\, X\, P'$, follows from the latter proposition and from the assumption "$P'\,\lceil v' \rceil$". The remaining of the proof is like in the case of non-polymorphic functions. □

Corollary 6.2.1 (Soundness for integer results) *Let $\hat{t}$ be a closed typed term of type int and let n be an integer. Let t denote the term obtained by stripping type annotations out of $\hat{t}$.*

$$\llbracket \hat{t} \rrbracket^{\emptyset}\ (= n) \qquad \Rightarrow \qquad t \Downarrow n$$

Proof Apply the soundness theorem with P instantiated as the predicate $(= n)$, and use the fact that typed reductions entail untyped reductions. □

6.2.2 Soundness of generated axioms

The proof of soundness of the generated axioms involves Hilbert's epsilon operator. Let me recall how it works. Given a predicate P of type $A \to \mathsf{Prop}$, where the type A is inhabited, Hilbert's epsilon operator returns a value of type A. This value satisfies the predicate P if there exists at least one value of type A satisfying P. Otherwise, if no value of type A satisfy P, then the value returned by the epsilon operator is unspecified.

Lemma 6.2.1 (Soundness of the axioms generated for values) *Consider a top-level Caml definition "*$\mathsf{let}\, x = \hat{t}^S$*". The generated axiom* X *and the axiom* $\mathsf{X\_cf}$ *admit a sound interpretation.*

Proof To start with, assume S is a monomorphic type. I realize the axiom X by picking a value V of type $[\![T]\!]$ such that the evaluation of $\hat{t}$ terminates and returns the encoding of V.

$$\mathsf{Definition}\ \mathsf{X}\ :\ [\![T]\!]\ :=\ \epsilon.\ \big(\lambda V.\ \hat{t} \Downarrow^{:} \lfloor V \rfloor\big).$$

Remark: the epsilon operator can be applied because the type $[\![T]\!]$ has been proved to be inhabited through the generated lemma $\mathsf{X\_safe}$. Note that if the term t does not terminate, then the value of X is unspecified.

It remains to justify the axiom $\mathsf{X\_cf}$. The goal is the proposition $\forall P.\, [\![\hat{t}]\!]^{\emptyset}\, P \Rightarrow P\,\mathsf{X}$. To prove this implication, consider a particular post-condition P and assume $[\![\hat{t}]\!]^{\emptyset}\, P$. The goal is to prove $P\,\mathsf{X}$. By the soundness theorem, $[\![\hat{t}]\!]^{\emptyset}\, P$ implies the existence of a value v such that $P\,\lceil v \rceil$ and $\hat{t} \Downarrow^{:} \hat{v}$. Thus, there exists at least one value V, namely $\lceil \hat{v} \rceil$, such that the proposition $\hat{t} \Downarrow^{:} \lfloor V \rfloor$ holds. Indeed, the value $\lfloor V \rfloor$ is equal to $\hat{v}$. As a consequence, the epsilon operator returns a value X such that the proposition $\hat{t} \Downarrow^{:} \lfloor X \rfloor$ holds. By determinacy of typed reductions, $\lfloor X \rfloor$ is equal to $\hat{v}$. So, X is equal to $\lceil \hat{v} \rceil$. The conclusion $P\,X$ is therefore derivable from $P\,\lceil \hat{v} \rceil$.

Generalization to polymorphic definitions The proof in this case generalizes the previous proof by adding quantification over polymorphic type variables. Let $\forall \overline{A}.T$ be the form of S. The type scheme $[\![S]\!]$ takes the form $\forall \overline{A}.[\![T]\!]$.

$$\mathsf{Definition}\ \mathsf{X}\ :\ [\![S]\!]\ :=\ \lambda \overline{A}.\ \epsilon.\ (\lambda (V : [\![T]\!]).\ \hat{t} \Downarrow^{:} \lfloor V \rfloor).$$

Recall that the value X is specified through monomorphic instances of a predicate P of type $\forall \overline{A}.([\![T]\!] \to \mathsf{Prop})$. To justify the axiom $\mathsf{X\_cf}$, the goal is to prove:

$$\forall P.\, \forall \overline{A}.\ [\![\hat{t}]\!]^{\emptyset}\, (P\,\overline{A}) \ \Rightarrow\ (P\,\overline{A})\,(\mathsf{X}\,\overline{A})$$

Consider some arbitrary types $\overline{\mathcal{T}}$, and let $\overline{T}$ denote $\|\overline{\mathcal{T}}\|$. The goal is to prove that the assumption "$[\![\, [A \to T]\, \hat{t}\,]\!]^{\emptyset}\, (P\, \overline{\mathcal{T}})$" implies the proposition "$(P\, \overline{\mathcal{T}})\, (\mathsf{X}\, \overline{\mathcal{T}})$".

The soundness theorem applied to the term $[\overline{A} \to \overline{T}]\, \hat{t}$ of type $[\overline{A} \to \overline{T}]\, T$ gives the existence of a value $\hat{v}$ of type $[\overline{A} \to \overline{T}]\, T$ such that $P\, \overline{\mathcal{T}}\, \lceil \hat{v} \rceil$ and $[\overline{A} \to \overline{T}]\, \hat{t} \Downarrow^{:} \hat{v}$. Therefore, there exists at least one value V, namely $\lceil \hat{v} \rceil$, such that the proposition $t \Downarrow^{:} \lfloor V \rfloor$ holds. As a consequence, the definition of $\mathsf{X}\, \overline{\mathcal{T}}$, shown below, yields a value V' such that the proposition $[\overline{A} \to \overline{T}]\, \hat{t} \Downarrow^{:} \lfloor V' \rfloor$ holds.

$$\mathsf{X}\, \overline{\mathcal{T}} \quad = \quad \epsilon.\, (\lambda(V : \lceil\!\lceil [\overline{A} \to \overline{T}]\, T \rceil\!\rceil).\ ([\overline{A} \to \overline{T}]\, \hat{t}) \Downarrow^{:} \lfloor V \rfloor)$$

Hence, $[\overline{A} \to \overline{T}]\, \hat{t} \Downarrow^{:} \lfloor \mathsf{X}\, \overline{\mathcal{T}} \rfloor$. By determinacy of typed reductions, $\lfloor \mathsf{X}\, \overline{\mathcal{T}} \rfloor$ is equal to $\hat{v}$. So $\mathsf{X}\, \overline{\mathcal{T}}$ is equal to $\lceil \hat{v} \rceil$. The conclusion $P\, \overline{\mathcal{T}}\, (\mathsf{X}\, \overline{\mathcal{T}})$ thus follows from the fact $P\, \overline{\mathcal{T}}\, \lceil \hat{v} \rceil$, which comes from the earlier application of the soundness theorem. □

Lemma 6.2.2 (Soundness of the axioms generated for functions) *Consider a top-level Caml function defined as follows "*$\mathsf{let\ rec}\ f = \Lambda\overline{A}.\lambda x_1 \ldots x_n.\hat{t}$*". The generated axiom* F *and the axiom* $\mathsf{F\_cf}$ *admit a sound interpretation.*

Proof I show that the generated axioms are consequences of the axioms generated for the equivalent definition "$\mathsf{let}\ f\ =\ (\mathsf{let\ rec}\ f\ =\ \Lambda\overline{A}.\lambda x_1 \ldots x_n.\hat{t}\ \mathsf{in}\ f)$", for which the previous soundness lemma applies. The goal is to prove the proposition "$\mathbf{body}_{\overline{A}}\ F\ X_1 \ldots X_n\ =\ [\![\, \hat{t}\,]\!]^{(f \mapsto \mathsf{F}, x_i \mapsto X_i)}$". To that end, I exploit the axiom generated for the declaration "$\mathsf{let}\ f\ =\ (\mathsf{let\ rec}\ f\ =\ \lambda x_1 \ldots x_n.\, \hat{t}\, \mathsf{in}\ f)$", which is the proposition "$\forall P.\ [\![\mathsf{let\ rec}\ f = \lambda x_1 \ldots x_n.\, \hat{t}\, \mathsf{in}\ f]\!]^{\emptyset}\, P \Rightarrow P\, F$". I apply this assumption with P instantiated as

$$\lambda G. \quad \mathbf{body}_{\overline{A}}\, G\, X_1 \ldots X_n\ =\ [\![\, \hat{t}\,]\!]^{(f \mapsto \mathsf{G}, x_i \mapsto X_i)}$$

It remains to prove the premise "$[\![\mathsf{let\ rec}\ f\ =\ \lambda x_1 \ldots x_n.\, \hat{t}\, \mathsf{in}\ f]\!]^{\emptyset}\, P$". By definition of the characteristic formula of a function definition, this proposition unfolds to "$\forall F'.\ (\mathbf{body}_{\overline{A}}\ F'\ X_1 \ldots X_n\ =\ [\![\, \hat{t}\,]\!]^{(f \mapsto \mathsf{F}', x_i \mapsto X_i)}) \Rightarrow\ P\ F'$", which, given the definition of P, can be reformulated as the tautology "$\forall F'.\ P\, F' \Rightarrow P\, F'$". □

6.3 Completeness

From a high-level point of view, the completeness theorem asserts that any correct program can be proved correct using characteristic formulae. More precisely, the theorem asserts that characteristic formulae are always precise and expressive enough to formally establish the behavior of any Caml program. For example, if a given Caml program computes an integer n, then one can prove that the characteristic formula of that program holds of the post-condition $(= n)$. More generally, the characteristic formula of a program always holds of the *most-general specification* of its output, a notion formalized in this section.

6.3.1 Labelling of function closures

The CFML library is built upon three axioms: the constant Func, the predicate AppEval, and a lemma asserting that AppEval is deterministic. In the proof of soundness, I have given a concrete interpretation of those axioms. However, to establish completeness, those axioms must remain abstract because the end-user cannot exploit any assumption about them. Since the interpretation of the type Func can no longer be used to relate values of type Func with the function closure they describe, I rely on a notion of *correct labelling* for relating values of type Func with the function closures they correspond to. The notions of labelling and of correct labelling are explained in what follows.

Consider a function definition ($\mathsf{let\,rec}\, f = \Lambda\overline{A}.\lambda x.\hat{t}\ \mathsf{in}\ \hat{t}'$). The body description associated with the definition of f is recalled next.

$$\forall\overline{A}.\ \forall X.\ \forall P.\ (\llbracket\, \hat{t}\,\rrbracket^{(x\mapsto X, f\mapsto F)}\ P) \ \Rightarrow\ \mathsf{AppReturns}\, F\ X\ P$$

This proposition is the only available assumption for reasoning about applications of the function f in the term $\hat{t}'$. The notion of labelling is used to relate this assumption, which is expressed in terms of a constant F of type Func, with the function closure $\mu f.\Lambda\overline{A}.\lambda x.\hat{t}$, which gets substituted in the continuation $\hat{t}'$ during the execution of the program. In short, I tag the function closure $\mu f.\Lambda\overline{A}.\lambda x.\hat{t}$ with the label F, writing $(\mu f.\Lambda\overline{A}.\lambda x.\hat{t})^{\{F\}}$ to denote this association, and I then define the decoding of the value $(\mu f.\Lambda\overline{A}.\lambda x.\hat{t})^{\{F\}}$ as the constant F. This labelling technique makes it possible to exploit the substitution lemma when reasoning on function definitions, through the following equality.

$$\llbracket\, \hat{t}'\,\rrbracket^{(f\mapsto F)} \quad = \quad \llbracket\, [f \to (\mu f.\Lambda\overline{A}.\lambda x.\hat{t})^{\{F\}}]\, \hat{t}'\,\rrbracket^{\emptyset}$$

Labelled terms are like typed terms except that every function closure is labelled with a value of type Func. I let $\tilde{t}$ stand for a copy of a typed term $\hat{t}$ in which all function closures are labelled. Similarly I use the notation $\tilde{v}$ for labelled values. The formal grammars are shown next.

Definition 6.3.1 (Labelled terms and values)

$$\begin{array}{lcl}
\tilde{v} & := & x\,\overline{T} \ \mid\ n \ \mid\ D\,\overline{T}(\tilde{v},\ldots,\tilde{v}) \ \mid\ (\mu f.\Lambda\overline{A}.\lambda x.\tilde{t})^{\{F\}} \\
\tilde{w} & := & \Lambda\overline{A}.\,\tilde{v} \\
\tilde{t} & := & \tilde{v} \ \mid\ \tilde{v}\,\tilde{v} \ \mid\ \mathit{crash} \ \mid\ \mathit{if}\,\tilde{v}\,\mathit{then}\,\tilde{t}\,\mathit{else}\,\tilde{t} \ \mid \\
 & & \mathit{let}\, x = \Lambda\overline{A}.\,\tilde{t}\ \mathit{in}\,\tilde{t} \ \mid\ \mathit{let\,rec}\, f = \Lambda\overline{A}.\lambda x.\tilde{t}\ \mathit{in}\,\tilde{t}
\end{array}$$

Given a labelled term $\tilde{t}$, one can recover the corresponding typed term $\hat{t}$ by stripping the labels out of function closures. Formally, the stripping function is called strip_labels, so the typed term $\hat{t}$ associated with $\tilde{t}$ is written strip_labels $\tilde{t}$.

The decoding of a labelled function closure is formally defined as shown below. Note that the decoding context Γ is ignored, since a function closure is always closed.

Definition 6.3.2 (Decoding of a labelled function) *Let* $(\mu f.\Lambda\overline{A}.\lambda x.\hat{t})^{\{F\}}$ *be a function closure labelled with a constant* F *be a constant of type* Func*, and let* Γ *be a decoding context.*

$$\lceil (\mu f.\Lambda\overline{A}.\lambda x.\tilde{t})^{\{F\}} \rceil^{\Gamma} \quad \equiv \quad F$$

The construction of characteristic formulae involves applications of the decoding operators. Since decoding now applies to labelled values, characteristic formulae need to be applied to labelled terms. So, in the proof of completeness, characteristic formulae take the form $[\![\, \tilde{t} \,]\!]$.

The central invariant of the proof of completeness is the notion of *correct labelling*. A function $(\mu f.\Lambda\overline{A}.\lambda x.\tilde{t})^{\{F\}}$ is said to be correctly labelled if the body description of that function, in which the function is being referred to as F, has been provided as assumption at a previous point in the characteristic formula. The notion of correct labelling is formalized through the definition of the predicate called body, as shown next.

Definition 6.3.3 (Correct labelling of a function)

$$\mathit{body}\ (\mu f.\Lambda\overline{A}.\lambda x.\hat{t})^{\{F\}} \ \equiv\ (\forall \overline{A}.\ \forall X.\ \forall P.\ [\![\, \tilde{t} \,]\!]^{(x \mapsto X, f \mapsto F)}\ P \ \Rightarrow\ \mathit{AppReturns}\ F\ X\ P)$$

A labelled term $\tilde{t}$ is then said to be correctly labelled if, for any function closure $(\mu f.\Lambda\overline{A}.\lambda x.\tilde{t}_1)^{\{F\}}$ that occurs in $\tilde{t}$, the proposition "$\mathsf{body}\,(\mu f.\Lambda\overline{A}.\lambda x.\tilde{t}_1)^{\{F\}}$" does hold. Moreover, the term $\tilde{t}$ is said to be *correctly labelled with values from a set* E when $\tilde{t}$ is correctly labelled and all the labels involved in $\tilde{t}$ belong to the set E. The ability to quantify over all the labels occurring in a labelled term is needed for the proofs. The correct labelling of $\tilde{t}$ with values from the set E is captured by the proposition "$\mathsf{labels}\ E\ \tilde{t}$", defined next.[2]

Definition 6.3.4 (Correct labelling of a term) *Let* $\tilde{t}$ *be a labelled term and let* E *be a set of Coq values of type* Func*.*

$$\mathit{labels}\ E\ \tilde{t} \quad \equiv \quad \left(\begin{array}{c} \textit{For any labelled function } (\mu f.\Lambda\overline{A}.\lambda x.\tilde{t}_1)^{\{F\}} \textit{ occuring in } \tilde{t}, \\ F \in E \quad \wedge \quad \mathit{body}\ (\mu f.\Lambda\overline{A}.\lambda x.\tilde{t}_1)^{\{F\}} \end{array} \right)$$

The same definition may be applied to values. For example, "$\mathsf{labels}\ E\ \tilde{v}$" indicates that the value $\tilde{v}$ is correctly labelled with constants from the set E.

A few immediate properties about the predicate labels are used in proofs. First, a source program $\tilde{t}$ does not contain any closure, so "$\mathsf{labels}\ \emptyset\ \tilde{t}$" holds. Second, "$\mathsf{labels}\ E\ \tilde{t}$" is preserved when the set E grows into a bigger set. Third, if "$\mathsf{labels}\ E\ \tilde{t}$" holds then, for any subterm $\tilde{t}'$ of $\tilde{t}$, "$\mathsf{labels}\ E\ \tilde{t}'$" holds as well. Finally, the notion of correct labelling is stable through substitution: if "$\mathsf{labels}\ E\ \tilde{t}$" and "$\mathsf{labels}\ E\ \tilde{w}$" holds, then "$\mathsf{labels}\ E\ ([x \to \tilde{w}]\,\tilde{t})$" holds as well.

[2] The definition of the predicate labels is stated using a meta-level quantification of the form "for any labelled function occurring in $\tilde{t}$". Alternatively, this meta-level quantification could be replaced with an inductive definition that follows the structure of $\tilde{t}$.

6.3.2 Most-general specifications

The next step towards stating and proving the completeness theorem is the definition of the notion of *most-general specification of a value*. Characteristic formulae allow to specify the result $\hat{v}$ of the evaluation of a term $\hat{t}$ with utmost accuracy, except for the functions that are contained in $\hat{v}$. Indeed, the best knowledge that can be gathered about a function from a characteristic formula is the body description of that function. In particular, it is not possible to state properties about the source code of a function closure.

So, the most-general specification of a typed value $\hat{v}$ of type T is a predicate of type "$[\![T]\!] \to \mathsf{Prop}$", written $\mathsf{mgs}\,\hat{v}$, that describes the shape of the value $\hat{v}$, except for the functions it contains. Functions are instead described using the predicate labels introduced in the previous section. More precisely, the predicate "$\mathsf{mgs}\,\hat{v}$" holds of a Coq value V if there exist a set E of Coq values of type Func and a labelling $\tilde{v}$ of $\hat{v}$ such that $\tilde{v}$ is correctly labelled with constants from E and V is equal to the decoding of $\lceil\tilde{v}\rceil$.

Definition 6.3.5 (Most-general specification of a value) *Let $\hat{v}$ be a well-typed value.*

$$\mathit{mgs}\,\hat{v} \quad \equiv \quad \lambda V.\ \exists E.\ \exists \tilde{v}.\ \hat{v} = \mathit{strip\_labels}\,\tilde{v} \ \wedge\ \mathit{labels}\,E\,\tilde{v} \ \wedge\ V = \lceil\tilde{v}\rceil$$

For example, the most general specification of the Caml value $(3, \mathsf{id}^{(\mathsf{int}\to\mathsf{int})})$, which is made of a pair of the integer 3 and of the identity function for integers $(\mu f.\Lambda.\lambda x^{\mathsf{int}}.x^{\mathsf{int}})$, is defined as:

$$\lambda(V : \mathsf{int}\times\mathsf{Func}).\ \exists E.\ \exists \tilde{v}.\ \mathsf{strip\_labels}\,\tilde{v} = (3, \mathsf{id}^{(\mathsf{int}\to\mathsf{int})}) \ \wedge\ \mathsf{labels}\,E\,\tilde{v} \ \wedge\ V = \lceil\tilde{v}\rceil$$

The labelled valued $\tilde{v}$ must be of the form $(3, \mathsf{id}^{(\mathsf{int}\to\mathsf{int})\{F\}})$ for some constant F that belongs to E. The assumption "$\mathsf{labels}\ E\ \tilde{v}$" is equivalent to $\mathsf{body}\,(\mu f.\lambda x^{\mathsf{int}}.x^{\mathsf{int}})^{\{F\}}$. Since the set E need not contain any constant other than F, the most-general specification of the value $(3, \mathsf{id}^{(\mathsf{int}\to\mathsf{int})})$ can be simplified by quantifying directly on F, as shown next.

$$\lambda(V : \mathsf{int}\times\mathsf{Func}).\quad \exists(F : \mathsf{Func}).\quad \mathsf{body}\,(\mu f.\lambda x^{\mathsf{int}}.x^{\mathsf{int}})^{\{F\}} \quad\wedge\quad V = (3, F)$$

Thus, the most-general specification of the Caml value $(3, \mathsf{id})$ holds of a Coq value V if and only if V is of the form $(3, F)$ for some constant F of type Func that admits the body description of the identity function for integers. Note that F is *not* specified as being equal to be the identity function. In fact, the statement "F is equal to $(\mu f.\Lambda.\lambda x^{\mathsf{int}}.x^{\mathsf{int}})$" is not even well-formed, because Func is here viewed as an abstract data type. The value F is only specified extensionally, through the predicate $\mathsf{AppReturns}$ involved in the definition of the predicate body.

6.3.3 Completeness theorem

Theorem 6.3.1 (Completeness in terms of most-general specifications) *Let $\hat{t}$ be a closed, well-typed term of type T that does not contain any function closure,*

and let $\hat{v}$ be a value.

$$\hat{t} \Downarrow^{:} \hat{v} \quad \Rightarrow \quad [\![\, \hat{t} \,]\!]^{\emptyset}\, (\mathit{mgs}\, \hat{v})$$

Proof I prove by induction on the derivation of the typed reduction judgment $\hat{t} \Downarrow^{:} \hat{v}$ that, for any correct labelling $\tilde{t}$ of $\hat{t}$ using constants for some set E, under the assumption that $\hat{t}$ is well-typed, the characteristic formula of $\tilde{t}$ holds of the most-general specification of v.

$$\hat{t} \Downarrow^{:} \hat{v} \quad \Rightarrow \quad \forall E.\ \forall \tilde{t}.\ \hat{t} = \mathsf{strip\_labels}\, \tilde{t} \ \wedge\ \mathsf{labels}\ E\ \tilde{t} \ \wedge\ (\vdash \hat{t}) \ \Rightarrow\ [\![\, \tilde{t} \,]\!]^{\emptyset}\, (\mathsf{mgs}\, \hat{v})$$

There is one proof case for each possible reduction rule. In each proof case, I describe the form of the term $\hat{t}$ and the premises of the assumption $\hat{t} \Downarrow^{:} \hat{v}$.

• Case $\hat{t} = \hat{v}$. By hypothesis, there exists a labelling $\tilde{t}$ of the term $\hat{t}$ such that "$\mathsf{labels}\ E\ \tilde{t}$" holds. Since $\hat{t}$ is equal to $\hat{v}$, there also exists a labelling $\tilde{v}$ of the value $\hat{v}$ such that "$\mathsf{labels}\ E\ \tilde{v}$" holds. The goal is $(\mathsf{mgs}\, \hat{v})\, \lceil \tilde{v} \rceil$. By definition, this proposition is equivalent to the following proposition, which is true.

$$\exists E.\ \exists \tilde{v}.\ \ \mathsf{strip\_labels}\, \tilde{v} = \hat{v} \ \wedge\ \mathsf{labels}\ E\ \tilde{v} \ \wedge\ \lceil \tilde{v} \rceil = \lceil \tilde{v} \rceil$$

• Case $\hat{t} = (\mathsf{let\,rec}\, f = \Lambda \overline{A}.\lambda x.\hat{t}_1\ \mathsf{in}\ \hat{t}_2)$, with a premise asserting that the term "$[f \rightarrow \mu f.\Lambda \overline{A}.\lambda x.\hat{t}_1]\, \hat{t}_2$" reduces to $\hat{v}$. By hypothesis, there exists a set E, a labelling $\tilde{t}_1$ of $\hat{t}_1$ and a labelling $\tilde{t}_2$ of $\hat{t}_2$ such that the propositions "$\mathsf{labels}\ E\ \tilde{t}_1$" and "$\mathsf{labels}\ E\ \tilde{t}_2$" hold. The proof obligation is "$\forall F.\ \mathcal{H} \ \Rightarrow\ [\![\tilde{t}_2]\!]^{(f \mapsto F)}\, (\mathsf{mgs}\, \hat{v})$", where $\mathcal{H}$, the body description of the function, corresponds exactly to the proposition $\mathsf{body}\, (\mu f.\Lambda \overline{A}.\lambda x.\tilde{t}_1)^{\{F\}}$.

Given a particular variable F of type Func, the goal simplifies to

$$\mathsf{body}\, (\mu f.\Lambda \overline{A}.\lambda x.\tilde{t}_1)^{\{F\}} \quad \Rightarrow \quad [\![\tilde{t}_2]\!]^{(f \mapsto F)}\, (\mathsf{mgs}\, \hat{v})$$

Since the decoding of $(\mu f.\Lambda \overline{A}.\lambda x.\tilde{t}_1)^{\{F\}}$ is equal to F, the substitution lemma (adapted to labelled terms) allows us to rewrite the formula $[\![\tilde{t}_2]\!]^{(f \mapsto F)}$ into $[\![[f \rightarrow (\mu f.\Lambda \overline{A}.\lambda x.\tilde{t}_1)^{\{F\}}]\, \tilde{t}_2]\!]^{\emptyset}$.

It remains to invoke the induction hypothesis on the reduction sequence $[f \rightarrow \mu f.\Lambda \overline{A}.\lambda x.\hat{t}_1]\, \hat{t}_2 \Downarrow^{:} \hat{v}$. To that end, it suffices to check that $[f \rightarrow (\mu f.\Lambda \overline{A}.\lambda x.\tilde{t}_1)^{\{F\}}]\, \tilde{t}_2$ is correctly labelled with values from the set $E \cup \{F\}$. Indeed, a label occurring in the term $[f \rightarrow (\mu f.\Lambda \overline{A}.\lambda x.\tilde{t}_1)^{\{F\}}]\, \tilde{t}_2$ either occurs in $\tilde{t}_1$, which is labelled with constants from E, or occurs in $\tilde{t}_2$, which is also labelled with constants from E, or is equal to the label F.

• Case $\hat{t} = ((\mu f.\Lambda \overline{A}.\lambda x^{T_0}.\hat{t}_1{}^{T_1})\, (\hat{v}_2{}^{T_2}))^{T}$, with a premise asserting that the term "$[f \rightarrow \mu f.\Lambda A.\lambda x.\hat{t}_1]\, [x \rightarrow \hat{v}_2]\, [\overline{A} \rightarrow \overline{T}]\, \hat{t}_1$" reduces to $\hat{v}$, and two premises relating types: $T_2 = [\overline{A} \rightarrow \overline{T}]\, T_0$ and $T = [\overline{A} \rightarrow \overline{T}]\, T_1$. Let $\tilde{t}$ be the labelling of $\hat{t}$. The hypothesis "$\mathsf{labels}\ E\ \tilde{t}$" implies that there exist a set E, a value F from that set, and two labellings $\tilde{t}_1$ and $\tilde{v}_2$ such that the following facts are true: $(\mu f.\Lambda \overline{A}.\lambda x.\tilde{t}_1)$ is labelled with F in $\tilde{t}$, the proposition $\mathsf{body}\, (\mu f.\Lambda \overline{A}.\lambda x.\tilde{t}_1)^{\{F\}}$ holds, "$\mathsf{labels}\ E\ \tilde{t}_1$" holds, and "$\mathsf{labels}\ E\ \tilde{v}_2$" holds.

Since the decoding of the labelled value $(\mu f.\Lambda\overline{A}.\lambda x.\hat{t}_1)^{\{F\}}$ is the constant F, the goal $[\![\,\tilde{t}\,]\!]^{\emptyset}\,(\mathsf{mgs}\,\tilde{v})$ is equivalent to "$\mathsf{AppReturns}\,F\,\lceil\tilde{v}_2\rceil\,(\mathsf{mgs}\,\hat{v})$". To prove this goal, I apply the assumption $\mathsf{body}\,(\mu f.\Lambda\overline{A}.\lambda x.\tilde{t}_1)^{\{F\}}$, which asserts the following proposition.

$$\forall\overline{A}.\,\forall X.\,\forall P.\,([\![\tilde{t}_1]\!]^{(x\mapsto X, f\mapsto F)}\;P)\;\Rightarrow\;\mathsf{AppReturns}\,F\,X\,P$$

The types $\overline{A}$ are instantiated as $\lceil\overline{T}\rceil$, X is instantiated as $\lceil\tilde{v}_2\rceil$ and P is instantiated as $(\mathsf{mgs}\,v)$. It remains to prove the premise, which is the proposition "$[\![[\overline{A}\to\overline{T}]\,\tilde{t}_1]\!]^{(x\mapsto\lceil\tilde{v}_2\rceil, f\mapsto F)}\;(\mathsf{mgs}\,\hat{v})$". Through the substitution lemma, it becomes "$[\![[f\to(\mu f.\Lambda\overline{A}.\lambda x.\tilde{t}_1)^{\{F\}}]\,[x\to\tilde{v}_2]\,[\overline{A}\to\overline{T}]\,\tilde{t}_1]\!]^{\emptyset}\,(\mathsf{mgs}\,\hat{v})$". This proposition follows from the induction hypothesis applied to the typed term "$[f\to\mu f.\Lambda\overline{A}.\lambda x.\hat{t}_1]\,[x\to\hat{v}_2]\,[\overline{A}\to\overline{T}]\,\hat{t}_1$". This term is well-typed and admits the type T because the type of $\hat{v}_2$ is equal to $[\overline{A}\to\overline{T}]\,T_0$ and because T is equal to $[\overline{A}\to\overline{T}]\,T_1$. Note that the set of labels occurring in "$[f\to(\mu f.\Lambda\overline{A}.\lambda x.\tilde{t}_1)^{\{F\}}]\,[x\to\tilde{v}_2]\,[\overline{A}\to\overline{T}]\,\tilde{t}_1$" is included in the set of labels occurring in $\tilde{t}$, which is of the form $((\mu f.\Lambda\overline{A}.\lambda x.\tilde{t}_1)^{\{F\}}\,\tilde{v}_2)$.

• Case $\hat{t}=(\mathsf{if}\,\hat{v}\,\mathsf{then}\,\hat{t}_1\,\mathsf{else}\,\hat{t}_2)$. Because $\hat{t}$ is well-typed, the value $\hat{v}$ has type bool, so it is either equal to true or false. In both cases, the conclusion follows from the induction hypothesis.

• Case $\hat{t}=(\mathsf{let}\,x=\Lambda\overline{A}.\,\hat{t}_1\;\mathsf{in}\,\hat{t}_2)$, with two premises asserting "$\hat{t}_1\Downarrow^{:}\hat{v}_1$" and "$([x\to\Lambda\overline{A}.\,\hat{v}_1]\,\hat{t}_2)\Downarrow^{:}\hat{v}$". By hypothesis, there exist a set E, a labelling $\tilde{t}_1$ of $\hat{t}_1$ and a labelling $\tilde{t}_2$ of $\hat{t}_2$ such that the propositions "$\mathsf{labels}\;E\;\tilde{t}_1$" and "$\mathsf{labels}\;E\;\tilde{t}_2$" hold. Let me start with the case where x admits a monomorphic type, that is, assuming $\overline{A}$ to be empty. In this case, the goal is:

$$\exists P'.\quad [\![\tilde{t}_1]\!]^{\emptyset}\,P'\quad\wedge\quad(\forall X.\,P'\,X\;\Rightarrow\;[\![\tilde{t}_2]\!]^{(x\mapsto X)}\,(\mathsf{mgs}\,\hat{v}))$$

To prove this goal, I instantiate P' as the predicate $\mathsf{mgs}\,\hat{v}_1$. The first subgoal, "$[\![\tilde{t}_1]\!]^{\emptyset}\,(\mathsf{mgs}\,\hat{v}_1)$", follows from the induction hypothesis applied to the reduction "$\hat{t}_1\Downarrow^{:}\hat{v}_1$" and to the set E.

It remains to prove the second subgoal. Let T_1 be the type of the variable x, and let X be a value of type $\lceil T_1\rceil$ such that "$\mathsf{mgs}\,\hat{v}_1\,X$" holds. By definition of mgs, there exists a set E' and a labelling $\tilde{v}_1$ of $\hat{v}_1$ such that "$\mathsf{labels}\;E'\;\tilde{v}_1$" and "$X=\lceil\tilde{v}_1\rceil$". The goal is to prove $[\![\tilde{t}_2]\!]^{(x\mapsto X)}\,(\mathsf{mgs}\,\hat{v})$. By the substitution lemma, the formula "$[\![\tilde{t}_2]\!]^{(x\mapsto X)}$" is equivalent to "$[\![[x\to\tilde{v}_1]\,\tilde{t}_2]\!]^{\emptyset}$". So it remains to prove "$[\![[x\to\tilde{v}_1]\,\tilde{t}_2]\!]^{\emptyset}\;(\mathsf{mgs}\,\hat{v})$". This proposition follows from the induction hypothesis applied to the reduction derivation "$([x\to\hat{v}_1]\,\hat{t}_2)\Downarrow^{:}\hat{v}$" and to the $E\cup E'$, which we know correctly labels the term $[x\to\tilde{v}_1]\,\tilde{t}_2$ thanks to the hypotheses "$\mathsf{labels}\;E'\;\tilde{v}_1$" and "$\mathsf{labels}\;E\;\tilde{t}_2$".

Generalization to polymorphic let-bindings The variable x admits the polymorphic type $\forall\overline{A}.\,T_1$. The goal is:

$$\exists P'.\;\begin{cases}\forall\overline{A}.\,[\![\,\tilde{t}_1\,]\!]^{\emptyset}\,(P'\,\overline{A})\\ \forall X.\,(\forall\overline{A}.\,(P'\,\overline{A}\,(X\,\overline{A})))\;\Rightarrow\;[\![\,\tilde{t}_2\,]\!]^{(x\mapsto X)}\,(\mathsf{mgs}\,\hat{v})\end{cases}$$

I instantiate P' as $\lambda\overline{A}.\,\mathsf{mgs}\,\hat{v}_1$, which admits the type $\forall\overline{A}.(\lceil T_1\rceil \to \mathsf{Prop})$, as expected. For the first subgoal, let $\overline{\mathcal{T}}$ be some arbitrary types and let $\overline{T}$ denote the list $\lfloor\overline{\mathcal{T}}\rfloor$. We have to prove $[\![\,[\overline{A} \to T]\,\tilde{t}_1\,]\!]^{\emptyset}\,(P'\,\overline{\mathcal{T}})$. The post-condition $P'\,\overline{\mathcal{T}}$ is equal to $[\overline{A} \to \overline{\mathcal{T}}]\,(\mathsf{mgs}\,\hat{v}_1)$. It is straightforward to show that this proposition is equal to $\mathsf{mgs}\,([\overline{A} \to \overline{T}]\,\hat{v}_1)$. The proof obligation $[\![\,[\overline{A} \to \overline{T}]\,\tilde{t}_1\,]\!]^{\emptyset}\,(\mathsf{mgs}\,([\overline{A} \to \overline{T}]\,\hat{v}_1))$, is obtained by applying the induction hypothesis to "$([\overline{A} \to \overline{T}]\,\hat{t}_1) \Downarrow^{:} ([\overline{A} \to \overline{T}]\,\hat{v}_1)$", which is a direct consequence of the hypothesis "$\hat{t}_1 \Downarrow^{:} \hat{v}_1$".

The second proof obligation is harder to establish. Let X be a Coq value of type $\forall\overline{A}.\,\lceil T_1\rceil$ that satisfies the proposition $\forall\overline{A}.\,(P'\,\overline{A}\,(X\,\overline{A}))$. The goal is to prove $[\![\,\tilde{t}_2\,]\!]^{(x\mapsto X)}\,(\mathsf{mgs}\,\hat{v})$. The difficulty comes from the fact that we do not have an assumption about the polymorphic value X, but instead have different assumptions for every monomorphic instances of X. For this reason, the proof involves reasoning separately on each of the occurrences of the polymorphic variable x in the term $\tilde{t}_2$. So, let $\tilde{t}'_2$ be a copy of $\tilde{t}_2$ in which the occurrences of x have been renamed using a finite set of names x_i, such that each x_i occurs exactly once in $\tilde{t}'_2$. The goal becomes "$[\![\,\tilde{t}'_2\,]\!]^{(x_1\mapsto X,\ldots,x_n\mapsto X)}\,(\mathsf{mgs}\,\hat{v})$". The plan is to build an family of labellings $(\tilde{v}_1^i)_{i\in[1..n]}$ for the value $\hat{v}_1$ such that the goal can be rewritten in the form "$[\![[x_i \to \tilde{v}_1^i]\,\tilde{t}'_2]\!]^{\emptyset}\,(\mathsf{mgs}\,\hat{v})$".

Consider a particular occurrence of the variable x_i in $\tilde{t}'_2$. It takes the form of a type application "$x_i\,\overline{T}$", for some types $\overline{T}$. The decoding of this occurrence gives "$X\,\overline{T}$". Let $\overline{\mathcal{T}}$ stand for the list of Coq types $\lceil\overline{T}\rceil$. Instantiating the hypothesis about X on the types $\overline{\mathcal{T}}$ gives $(P'\,\overline{\mathcal{T}})\,(X\,\overline{\mathcal{T}})$. By definition of P', the value $X\,\overline{\mathcal{T}}$ thus satisfies the predicate $[\overline{A} \to \overline{\mathcal{T}}]\,(\mathsf{mgs}\,\hat{v}_1)$, which is the same as $\mathsf{mgs}\,([\overline{A} \to \overline{T}]\,\hat{v}_1)$. So, there exist a set E_i and a labelling $\tilde{v}_1^i$ of the value $\hat{v}_1$ such that "$\mathsf{labels}\ E_i\ \tilde{v}_1^i$" holds and such that $X\,\overline{\mathcal{T}}$ is equal to $\lceil\tilde{v}_1^i\rceil$.

To summarize the reasoning of the previous paragraph, the decoding of the occurrence of the variable x_i in the context $(x_1 \mapsto X, \ldots, x_n \mapsto X)$ is the same as the decoding of the labelled value $\tilde{v}_1^i$ in the empty context. It follows that the characteristic formula "$[\![\,\tilde{t}'_2\,]\!]^{(x_1\mapsto X,\ldots,x_n\mapsto X)}$" can be rewritten into "$[\![[x_i \to \tilde{v}_1^i]\,\tilde{t}'_2]\!]^{\emptyset}$". Furthermore, there exists a family of sets $(E_i)_{i\in[1..n]}$ such that "$\mathsf{labels}\ E_i\ \tilde{v}_1^i$" holds for every index i.

It remains to prove "$[\![[x_i \to \tilde{v}_1^i]\,\tilde{t}'_2]\!]^{\emptyset}\,(\mathsf{mgs}\,\hat{v})$". This conclusion follows from the induction hypothesis applied to the term $[x \to \Lambda\overline{A}.\,\hat{v}_1]\,\hat{t}_2$ and to the set $E \cup (\bigcup_i E_i)$. Indeed, the term $[x_i \to \Lambda\overline{A}.\,\tilde{v}_1^i]\,\tilde{t}'_2$ is a labelling of the term $[x \to \Lambda\overline{A}.\,\hat{v}_1]\,\hat{t}_2$. Also, a label that occurs in the labelled term $[x_i \to \tilde{v}_1^i]\,\tilde{t}'_2$ belongs to the set $E \cup (\bigcup_i E_i)$. Indeed such a label occurs either in $\tilde{t}_2$ (which contains the same labels as $\tilde{t}'_2$) or in one of the labelled values $\tilde{v}_1^i$, and we have "$\mathsf{labels}\ E\ \tilde{t}_2$" as well as "$\mathsf{labels}\ E_i\ \tilde{v}_1^i$" for every index i. □

6.3.4 Completeness for integer results

A simpler statement of the completeness theorem can be devised for a program that produces an integer. If a term t evaluates to an integer n that satisfies a predicate P, then the characteristic formula of t holds of P.

Corollary 6.3.1 (Completeness for integer results) *Consider a program well-typed in ML and that admits the type* int. *Let* $\hat{t}$ *be the term obtained by replacing ML type annotations with the corresponding weak-ML type annotations in that program, and let* t *denote the corresponding raw term. Let* n *be an integer and let* P *be a predicate on integers. Then,*

$$t \Downarrow n \quad \wedge \quad P\,n \quad \Rightarrow \quad [\![\,\hat{t}\,]\!]^{\emptyset}\; P$$

Proof The reduction $t \Downarrow n$ starts on a term well-typed in ML, So there exists a corresponding typed reduction $\hat{t} \Downarrow^{:} n$. The completeness theorem applied to this assumption gives "$[\![\,\hat{t}\,]\!]^{\emptyset}\ (\mathsf{mgs}\,n)$". To show "$[\![\,\hat{t}\,]\!]^{\emptyset}\ P$", I apply the weakening lemma for characteristic formulae. I remains to show that, for any value V of type int, the proposition "$\mathsf{mgs}\,n\,V$" implies "$P\,V$". By definition of mgs, there exists a set E and a labelling $\tilde{n}$ of n such that $\mathsf{labels}\ E\ \tilde{n}$ and $V = n$. Since n is an integer, the labelling can be ignored completely. So, $V = \lceil n \rceil$ and the conclusion "$P\,V$" then follows from the hypothesis $P\,n$. □

6.4 Quantification over Type

Polymorphism has been treated by quantifying over logical type variables. I have not yet discussed what exactly is the sort of these variables in the logic. A tempting solution would be to assign them the sort Type. In Coq, Type is the sort of all types from the logic. However, type variables used to represent polymorphism are only meant to range over reflected types, i.e. types of the form $[\![T]\!]$. Thus, it seems that one ought to assign type variables the sort RType, defined as $\{\, A : \mathsf{Type} \mid \exists T.\, A = [\![T]\!] \,\}$.

Of course, I would provide RType as an abstract definition, since the fact that types correspond to reflected types need not be exploited in proofs. A question naturally arises: since RType is an abstract type, would it remain sound and complete to use the sort Type instead of the sort RType as a sort for type variables? The answer is positive. The purpose of this section is to justify this claim.

6.4.1 Case study: the identity function

To start with, I consider an example suggesting how replacing the quantification over RType with a quantification over Type affects characteristic formulae. Consider the identity function id, defined as $\lambda x.\, x$. This function is polymorphic: it admits the type "$\forall A.\, A \to A$" in Caml. Hence, the assumption provided by a characteristic formula for this function involves a universal quantification over the type A.

$$\forall (A : \mathsf{RType}).\ \forall (X : A).\ \forall (P : A \to \mathsf{Prop}).\ P\,X \ \Rightarrow\ \mathsf{AppReturns}\,\mathsf{id}\,X\,P$$

For the sake of studying this formula, let me reformulate it into the following equivalent proposition[3], which is expressed in terms of the predicate AppEval instead of the high-level predicate AppReturns.

$$\forall(A : \mathsf{RType}).\ \forall(X : A).\ \mathsf{AppEval}\,\mathsf{id}\,X\,X$$

Is this statement still sound if we change the sort of A from RType to Type?

Let me first recall why the statement with RType is sound. Let A be a type of sort RType and let X be a Coq value of type A. The goal is to prove $\mathsf{AppEval}\,\mathsf{id}\,X\,X$. By definition of AppEval, it is equivalent to the proposition $(\lfloor\mathsf{id}\rfloor\,\lfloor X\rfloor) \Downarrow^{:} \lfloor X\rfloor$. The encoding of X, written $\lfloor A\rfloor$, is a well-typed program value $\hat{v}$. Thus, $\lfloor X\rfloor$ is equal to v. The goal then simplifies to $((\lambda x.\, x)\,\hat{v}) \Downarrow^{:} \hat{v}$, which is correct according to the semantics.

Now, if A is of sort Type and X is an arbitrary Coq value of type A, then there does not necessarily exist a program value $\hat{v}$ that corresponds to X. Intuitively, however, it would make sense to say that applying the identify function to some exotic object returns the exact same object. The object in question is not a real program value, but this is not a problem because it is never inspected in any way by the polymorphic identity function. The purpose of the next section is to formally justify this intuition.

6.4.2 Formalization of exotic values

I extend the grammar of program values with *exotic values*, in such a way as to obtain a bijection between the set of all Coq values and the set of all well-typed program values, for a suitably-defined notion of typing of exotic values. First, the grammar of values is extended with exotic values, written "$\mathsf{exo}\,\mathbb{T}\,V$", where $\mathbb{T}$ stands for a Coq type and V is a Coq value of type $\mathbb{T}$. Note that $\mathbb{T}$ ranges over all Coq types, whereas $\mathcal{T}$ only ranges over Coq types of the form $\llbracket T \rrbracket$. The grammar of weak-ML types is extended with exotic types, written "$\mathsf{Exotic}\,\mathbb{T}$".

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

A Coq type $\mathbb{T}$ is said to be exotic, written "$\mathsf{is\_exotic}\,\mathbb{T}$", if the head constructor of A does not correspond to a type constructor that exists in weak-ML. Let $\mathscr{C}$ denote the set of type constructors introduced through algebraic data type definitions. The definition of is_exotic is formalized as follows.[4]

$$\mathsf{is\_exotic}\,\mathbb{T} \quad \equiv \quad \begin{cases} \mathbb{T} \neq \mathsf{Int} \\ \mathbb{T} \neq \mathsf{Func} \\ \forall C \in \mathscr{C}.\ \forall \overline{\mathbb{T}}.\ \ \mathbb{T} \neq C\,\overline{\mathbb{T}} \end{cases}$$

[3] Given the definition of AppReturns, the equivalence to be established is

$$\mathsf{AppEval}\,\mathsf{id}\,X\,X \iff (\forall P.\, P\,X \Rightarrow (\exists Y.\, \mathsf{AppEval}\,\mathsf{id}\,X\,Y \wedge P\,Y))$$

From left to right, instantiate Y as X. From right to left, instantiate P as $\lambda Y.\,(Y = X)$.

[4] Technically, one should also check that the type constructors are applied to list of types of the appropriate arity. Partially-applied type constructors are also treated as exotic values.

An exotic value "$\mathsf{exo}\,\mathbb{T}\,V$" admits the type "$\mathsf{Exotic}\,\mathbb{T}$" if and only if the Coq type $\mathbb{T}$ is exotic.

$$\frac{\mathsf{is\_exotic}\,\mathbb{T}}{\Delta \vdash (\mathsf{exo}\,\mathbb{T}\,V)^{(\mathsf{Exotic}\,\mathbb{T})}}$$

Let $\mathbb{T}$ be an exotic Coq type. The programming language type "$\mathsf{Exotic}\,\mathbb{T}$" is reflected in the logic as the type $\mathbb{T}$. The decoding of an exotic value "$\mathsf{exo}\,\mathbb{T}\,V$" is V, and, reciprocally, the encoding of a Coq value V of some exotic type $\mathbb{T}$ is the exotic value "$\mathsf{exo}\,\mathbb{T}\,V$".

$$\left.\begin{array}{lcl} \lceil\!\lceil \mathsf{Exotic}\,\mathbb{T} \rceil\!\rceil & \equiv & \mathbb{T} \\ \lfloor\!\lfloor \mathbb{T} \rfloor\!\rfloor & \equiv & \mathsf{Exotic}\,\mathbb{T} \\ \lceil \mathsf{exo}\,\mathbb{T}\,V \rceil & \equiv & V \\ \lfloor V : \mathbb{T} \rfloor & \equiv & \mathsf{exo}\,\mathbb{T}\,V \end{array}\right\} \textit{when ``is\_exotic}\,\mathbb{T}\textit{'' holds}$$

Observe that the side-condition "$\mathsf{is\_exotic}\,\mathbb{T}$" ensures that the definition of $\lfloor\!\lfloor \mathbb{T} \rfloor\!\rfloor$ and of $\lfloor V : \mathbb{T} \rfloor$ do not overlap with existing definitions. The soundness of the introduction of exotic values is captured through the following lemma.

Lemma 6.4.1 (Inverse functions with exotic values) *Under the definitions extended with exotic types and exotic values, the operator* $\lfloor\!\lfloor \cdot \rfloor\!\rfloor$ *is the inverse of* $\lceil\!\lceil \cdot \rceil\!\rceil$ *and* $\lfloor \cdot \rfloor$ *is the inverse of* $\lceil \cdot \rceil$.

Proof Follows directly from the definitions of the operations on exotic types and exotic values. □

To summarize, the treatment of exotic values described in this section allows for a bijection to be established between the set of all Coq types and the set of all weak-ML types, and between the set of all Coq values and the set of all typed program values. This bijection justifies the quantification over the sort Type of the type variables that occur in characteristic formulae.

Chapter 7

Proofs for the imperative setting

Through the previous chapter, I have established the soundness and completeness of characteristic formulae for pure programs. The matter of this chapter is the generalization of those results to the case of imperative programs. To that end, the theorems need to be extended with heap descriptions. Two additional difficulties related to the treatment of mutable state arise. Firstly, the statement of the soundness theorem needs to quantify over the pieces of heap that have been framed out before reasoning on a given characteristic formula. Secondly, the statement of the completeness theorem needs to account for the fact that characteristic formulae do not reveal the actual address of the memory cells being allocated. The most general post-condition that can be proved from a characteristic formula is thus a specification of the output value and of the output heap up to renaming of locations.

7.1 Additional definitions and lemmas

7.1.1 Typing and translation of memory stores

Reasoning on characteristic formulae involves the notion of typed memory store, written $\hat{m}$, and that of well-typed memory stores. I define those notions next, and then describe the bijection between the set of well-typed memory stores and the set of values of type Heap.

A memory store m is a map from locations to closed values. A typed store $\hat{m}$ is a map from locations to typed values. So, if l is a location in the domain of $\hat{m}$, then $\hat{m}[l]$ denotes the typed value stored in $\hat{m}$ at that location. A typed store $\hat{m}$ is said to be well-typed in weak-ML, written $\vdash \hat{m}$, if all the values stored in $\hat{m}$ are well-typed. Formally,

$$(\vdash \hat{m}) \qquad \equiv \qquad \forall l \in \mathrm{dom}(\hat{m}). \quad (\vdash \hat{m}[l])$$

A well-typed memory store $\hat{m}$ can be decoded into a value of type Heap. To that end, it suffices to decode each of the values stored in $\hat{m}$. Reciprocally, a value of

type Heap can be encoded into a well-typed memory store by encoding all the values that it contains. The formalization of this definition is slightly complicated by the fact that Heap is actually a map from locations to dependent pairs made of a Coq type and of a Coq value of that type. The formal definition is shown next. Note that the decoding of typed stores only applies to well-typed stores.

Definition 7.1.1 (Decoding and encoding of memory stores)

$$\lceil \hat{m} \rceil \equiv h \quad \text{where} \begin{cases} \text{dom}(h) = \text{dom}(\hat{m}) \\ \hat{m}[l] = \hat{v}^T \;\Rightarrow\; h[l] = (\llbracket T \rrbracket,\ \lceil \hat{v}^T \rceil) \end{cases}$$

$$\lfloor h \rfloor \equiv \hat{m} \quad \text{where} \begin{cases} \text{dom}(\hat{m}) = \text{dom}(h) \\ h[l] = (\mathcal{T},\ V : \mathcal{T}) \;\Rightarrow\; \hat{m}[l] = \lfloor V \rfloor \end{cases}$$

It is straightforward to check that $\lfloor \lceil \hat{m} \rceil \rfloor$ is equal to $\hat{m}$ for any well-typed memory store $\hat{m}$, and that $\lceil \lfloor h \rfloor \rceil$ is equal to h for any heap h.

Remark: for the presentation to be complete, the decoding and the encoding operations needs to be extended to locations, as follows.

$$\begin{array}{lcllcl} \llbracket \mathsf{loc} \rrbracket & \equiv & \mathsf{Loc} & \lceil l^{\mathsf{loc}} \rceil & \equiv & (l : \mathsf{Loc}) \\ \llbracket \mathsf{Loc} \rrbracket & \equiv & \mathsf{loc} & \lfloor l : \mathsf{Loc} \rfloor & \equiv & l^{\mathsf{loc}} \end{array}$$

7.1.2 Typed reduction judgment

The typed version of the reduction judgment is defined in a similar manner as in the purely functional setting, except that it now involves typed memory stores. The judgment takes the form $\hat{t}_{/\hat{m}} \Downarrow^{:} \hat{v}'_{/\hat{m}'}$, asserting that the evaluation of the typed $\hat{t}$ in the typed store $\hat{m}$ terminates and returns the typed value $\hat{v}'$ in the typed store $\hat{m}'$. The inductive rules appear in Figure 7.1. Observe that the semantics of the primitive functions ref, get and set consists in reading and writing *typed* values in a typed store. The reduction rule for get includes a premise asserting that the value at location l in the store has the same type as the term "get l".

As before, the typed reduction judgment is deterministic and preserves typing.

Lemma 7.1.1 (Determinacy of typed reductions)

$$\hat{t}_{/\hat{m}} \Downarrow^{:} \hat{v}'_{1/\hat{m}'_1} \quad \wedge \quad \hat{t}_{/\hat{m}} \Downarrow^{:} \hat{v}'_{2/\hat{m}'_2} \quad \Rightarrow \quad \hat{v}'_1 = \hat{v}'_2 \quad \wedge \quad \hat{m}'_1 = \hat{m}'_2$$

Proof Similar proof as in the purely-functional setting (Lemma 6.1.7). □

Lemma 7.1.2 (Typed reductions preserve typing) *Let $\hat{t}$ be a typed term, let $\hat{v}$ be a typed value, let $\hat{m}$ and $\hat{m}$ be two typed stores, let T be a type and let $\overline{B}$ denote a list of free type variables.*

$$\overline{B} \vdash \hat{t}^T \quad \wedge \quad \vdash \hat{m} \quad \wedge \quad \hat{t}_{/\hat{m}} \Downarrow^{:} \hat{v}_{/\hat{m}'} \quad \Rightarrow \quad \overline{B} \vdash \hat{v}^T \quad \wedge \quad \vdash \hat{m}'$$

$$\frac{}{\hat{v}_{/\hat{m}} \Downarrow^{:} \hat{v}_{/\hat{m}}} \qquad \frac{\hat{t}_{1/\hat{m}_1} \Downarrow^{:} \hat{v}_{1/\hat{m}_2} \qquad ([x \to \hat{v}_1]\,\hat{t}_2)_{/\hat{m}_2} \Downarrow^{:} \hat{v}_{/\hat{m}_3}}{(\mathsf{let}\, x = \hat{t}_1 \,\mathsf{in}\, \hat{t}_2)_{/\hat{m}_1} \Downarrow^{:} \hat{v}_{/\hat{m}_3}}$$

$$\frac{([x \to \hat{w}_1]\,\hat{t}_2)_{/\hat{m}} \Downarrow^{:} \hat{v}_{/\hat{m}'}}{(\mathsf{let}\, x = \hat{w}_1 \,\mathsf{in}\, \hat{t}_2)_{/\hat{m}} \Downarrow^{:} \hat{v}_{/\hat{m}'}} \qquad \frac{([f \to \mu f.\Lambda\overline{A}.\lambda x.\hat{t}_1]\,\hat{t}_2)_{/\hat{m}} \Downarrow^{:} \hat{v}_{/\hat{m}'}}{(\mathsf{let\,rec}\, f = \Lambda\overline{A}.\lambda x.\hat{t}_1 \,\mathsf{in}\, \hat{t}_2)_{/\hat{m}} \Downarrow^{:} \hat{v}_{/\hat{m}'}}$$

$$\frac{\begin{array}{c}T_2 = [\overline{A} \to \overline{T}]\,T_0 \qquad T = [\overline{A} \to \overline{T}]\,T_1 \\ ([f \to \mu f.\Lambda\overline{A}.\lambda x.\hat{t}_1]\,[x \to \hat{v}_2]\,[\overline{A} \to \overline{T}]\,\hat{t}_1)_{/\hat{m}} \Downarrow^{:} \hat{v}_{/\hat{m}'}\end{array}}{((\mu f.\Lambda\overline{A}.\lambda x^{T_0}.\hat{t}_1^{\,T_1})^{\mathsf{func}}\,(\hat{v}_2^{\,T_2}))^{T}{}_{/\hat{m}} \Downarrow^{:} \hat{v}_{/\hat{m}'}} \qquad \frac{l = 1 + \mathsf{max}(\mathrm{dom}(\hat{m}))}{(\mathsf{ref}\,\hat{v})^{\mathsf{loc}}{}_{/\hat{m}} \Downarrow^{:} (l^{\mathsf{loc}})_{/\hat{m} \uplus [l \mapsto \hat{v}]}}$$

$$\frac{l \in \mathrm{dom}(\hat{m}) \qquad \hat{v}^T = \hat{m}[l]}{(\mathsf{get}\,(l^{\mathsf{loc}}))^{T}{}_{/\hat{m}} \Downarrow^{:} \hat{v}^{T}{}_{/\hat{m}}} \qquad \frac{l \in \mathrm{dom}(\hat{m})}{(\mathsf{set}\,(l^{\mathsf{loc}})\,\hat{v})^{\mathsf{unit}}{}_{/\hat{m}} \Downarrow^{:} tt^{\mathsf{unit}}{}_{/(\hat{m}[l \mapsto \hat{v}])}}$$

Figure 7.1: Typed reduction judgment for imperative programs

Proof The proof is similar to that of the purely-functional setting (Lemma 6.1.6), except when the store is involved in the reduction.

• Case $(\mathsf{ref}\,\hat{v})^{\mathsf{loc}}{}_{/\hat{m}} \Downarrow^{:} (l^{\mathsf{loc}})_{/\hat{m} \uplus [l \mapsto \hat{v}]}$. By assumption, $\hat{v}$ and $\hat{m}$ are well-typed, so the extended store $\hat{m} \uplus [l \mapsto \hat{v}]$ is also well-typed.

• Case $(\mathsf{get}\,(l^{\mathsf{loc}}))^{T}{}_{/\hat{m}} \Downarrow^{:} \hat{v}^{T}{}_{/\hat{m}}$, where $\hat{v}^T = \hat{m}[l]$. From the premise $(\vdash \hat{m})$, we get $(\vdash \hat{m}[l])$. Since $\hat{v}^T$ is equal to $\hat{m}[l]$, the value $\hat{v}^T$ is well-typed in the empty context.

• Case $(\mathsf{set}\,(l^{\mathsf{loc}})\,\hat{v})^{\mathsf{unit}}{}_{/\hat{m}} \Downarrow^{:} tt^{\mathsf{unit}}{}_{/(\hat{m}[l \mapsto \hat{v}])}$ By assumption, $\hat{v}$ and $\hat{m}$ are well-typed, so the updated store $\hat{m}[l \mapsto \hat{v}]$ is also well-typed. □

To state the soundness and the completeness theorem, I introduce the convenient notation $\hat{t}_{/h} \Downarrow^{:} \hat{v}'_{/h'}$, which asserts that the term $\hat{t}$ in a store equal to the encoding of the heap h evaluates to a value $\hat{v}'$ in a store equal to the encoding of the heap h'.

Definition 7.1.2 (Typed reduction judgment on heaps)

$$(\hat{t}_{/h} \Downarrow^{:} \hat{v}'_{/h'}) \quad \equiv \quad (\hat{t}_{/\lfloor h \rfloor} \Downarrow^{:} \hat{v}'_{/\lfloor h' \rfloor})$$

7.1.3 Interpretation and properties of AppEval

The proposition $\mathsf{AppEval}\,F\,V\,h\,V'\,h'$ captures the fact that application of the encoding of F to the encoding of V in the store described by the heap h terminates and returns the encoding of V' in the store described by the heap h'. The type of AppEval is recalled next.

$$\mathsf{AppEval} \;:\; \forall A\,B.\,\mathsf{Func} \to A \to \mathsf{Heap} \to B \to \mathsf{Heap} \to \mathsf{Prop}$$

The axiom AppEval is interpreted as follows.

Definition 7.1.3 (Definition of AppEval) *Let F be a value of type Func, V be a value of type $\mathcal{T}$, V' be a value of type $\mathcal{T}'$, and h and h' be two heaps.*

$$\mathit{AppEval}_{\mathcal{T},\mathcal{T}'}\, F\, V\, h\, V'\, h \quad \equiv \quad (\lfloor F \rfloor\, \lfloor V \rfloor)_{/\lfloor h \rfloor} \Downarrow^{:} \lfloor V' \rfloor_{/\lfloor h' \rfloor}$$

Remark: one can also present this definition as an inductive rule, as follows.

$$\frac{(\hat{f}\,\hat{v})_{/\hat{m}} \Downarrow^{:} \hat{v}'_{/\hat{m}'}}{\mathsf{AppEval}\ \lceil \hat{f} \rceil\ \lceil \hat{v} \rceil\ \lceil \hat{m} \rceil \lceil \hat{v}' \rceil \lceil \hat{m}' \rceil}$$

The determinacy lemma associated with the definition of AppEval follows directly from the determinacy of typed reductions.

Lemma 7.1.3 (Determinacy) *For any types $\mathcal{T}$ and $\mathcal{T}'$, for any Coq value F of type Func, any Coq value V of type $\mathcal{T}$, any two Coq values V'_1 and V'_2 of type $\mathcal{T}'$, any heap h and any two heaps h'_1 and h'_2.*

$$\mathit{AppEval}\, F\, V\, h\, V'_1\, h'_1 \quad \wedge \quad \mathit{AppEval}\, F\, V\, h\, V'_2\, h'_2 \quad \Rightarrow \quad V'_1 = V'_2 \quad \wedge \quad h'_1 = h'_2$$

7.1.4 Elimination of n-ary functions

In the previous chapter, I have exploited the fact that the characteristic formula of a n-ary application is equivalent to the characteristic formula of the encoding of that application in terms of unary applications. I could thereby eliminate reference to Spec_n and work entirely in terms of $\mathsf{AppReturns}_1$. This result is partially broken by the introduction of the predicate AppPure which is used to handle partial applications. In short, when describing a function of two arguments, the predicate Spec_2 from the imperative setting captures a stronger property than a proposition stated only in terms of $\mathsf{AppReturns}_1$, because Spec_2 forbids any modification to the store on the application of the first argument.

The direct characteristic formula for imperative n-ary functions gives a stronger hypothesis about functions than a description in terms of unary functions, so it leads to a logically-weaker characteristic formula. So, some work is needed in the proof of the soundness theorem for justifying the correctness of the assumption given for n-ary functions. The following lemma, proved in Coq, explains how the body description for the function "$\mathsf{let\,rec}\ f = \lambda xy.\, t$", expressed in terms of Spec_2, can be related to a combination of AppPure and $\mathsf{AppReturns}_1$, which is quite similar to the body description associated with the function "$\mathsf{let\,rec}\ f = \lambda x.\, (\mathsf{let\,rec}\ g = \lambda y.\, t\ \mathsf{in}\ g)$".

Lemma 7.1.4 (Spec_2-to-AppPure-and-$\mathsf{AppReturns}_1$)

$$\begin{array}{ll} & (\forall K.\ \mathit{is\_spec}_2\, K \wedge (\forall x\, y.\ K\, x\, y\, (G\, x\, y)) \Rightarrow \mathit{Spec}_2\, f\, K) \\ \iff & (\forall x\, P.\ (\forall g.\ \mathcal{H} \Rightarrow P\, g) \Rightarrow \mathit{AppPure}\, f\, x\, P) \\ & \textit{where}\ \mathcal{H} \equiv (\forall y\, H'\, Q'.\ (G\, x\, y)\, H'\, Q' \Rightarrow \mathit{AppReturns}_1\, g\, y\, H'\, Q') \end{array}$$

However, there is no further complication in the completeness theorem, because what can be proved with a given assumption about functions can also be proved with a stronger assumption about functions. The following lemma states that the body description for the function "$\mathsf{let\,rec}\, f = \lambda xy.\, t$", is strictly stronger than that of "$\mathsf{let\,rec}\, f = \lambda x.\, (\mathsf{let\,rec}\, g = \lambda y.\, t\, \mathsf{in}\, g)$". Below, $\mathcal{H}$ stands for the same proposition as in the previous lemma.

Lemma 7.1.5 ($\mathsf{Spec_2}$-to-$\mathsf{AppReturns_1}$)

$$\begin{array}{rl} & (\forall K.\ \mathit{is\_spec}_2\, K \wedge (\forall x\, y.\ K\, x\, y\, (G\, x\, y)) \Rightarrow \mathit{Spec}_2\, f\, K) \\ \Rightarrow & (\forall x\, H\, Q.\ (\forall g.\ \mathcal{H} \Rightarrow H \rhd Q\, g) \Rightarrow \mathit{AppPure}\, f\, x\, H\, Q) \end{array}$$

In summary, the proofs in this chapter need only be concerned with unary functions except for one proof case in the soundness theorem, in which I prove the correctness of the body description "$\forall x\, P.\ (\forall g.\ \mathcal{H} \Rightarrow P\, g) \Rightarrow \mathsf{AppPure} f x P$", which describes a functions of arity two. The treatment of higher arities does not raise any further difficulty.

7.2 Soundness

A simple statement of the soundness theorem can be given for complete executions. Assume that a term $\hat{t}$ is executed in an empty memory store and that its characteristic formula satisfies a post-condition Q. Then, the term $\hat{t}$ produces a value $\hat{v}$ and a heap of the form $h + h'$ such that the proposition "$Q\, \lceil \hat{v} \rceil\, h$" holds. The heap h' corresponds to the data that has been allocated and then discarded during the reasoning.

Theorem 7.2.1 (Soundness for complete executions) *Let $\hat{t}$ be a term carrying weak-ML type annotations, let T be the type of $\hat{t}$, and let Q be a post-condition of type $[\![T]\!] \to \mathit{Hprop}$.*

$$\vdash \hat{t} \quad \wedge \quad [\![\hat{t}]\!]^{\emptyset}\, [\,]\, Q \quad \Rightarrow \quad \exists \hat{v}\, h\, h'.\ \ \hat{t}_{/\varnothing} \Downarrow^{:} \hat{v}_{/(h+h')} \quad \wedge \quad Q\, \lceil \hat{v} \rceil\, h$$

Remark: the assumption $\vdash \hat{t}$ and the conclusion $\hat{t}_{/\varnothing} \Downarrow^{:} \hat{v}_{/(h+h')}$ ensure that the output value $\hat{v}$ is well-typed and admits the same type as $\hat{t}$.

Here again, representation predicates are defined in Coq and their properties are proved in Coq, so they are not involved at any point in the soundness proof. Indeed, all heap predicates of the form $x \rightsquigarrow S\, X$ are ultimately defined in terms of heap predicates of the form $l \hookrightarrow v$. Only the latter appear in the proof of soundness, which follows.

The general statement of soundness, upon which the induction is conducted, relies on an intermediate definition. The proposition "$\mathsf{sound}\ \hat{t}\, \mathcal{F}$" asserts that the formula $\mathcal{F}$ is a correct description of the behavior of the term $\hat{t}$. The definition of the predicate "sound" involves a reduction sequence of the form $\hat{t}_{/h_i+h_k} \Downarrow^{:} \hat{v}_{/h_f+h_k+h_g}$, where h_i and h_f correspond to the initial and final heap which are involved in the

specification of the term $\hat{t}$, where h_k corresponds to the pieces of heap that have been framed out, and where h_g corresponds to the piece of heap being discarded during the reasoning on the execution of the term $\hat{t}$.

Definition 7.2.1 (Soundness of a formula) *Let $\hat{t}$ be a well-typed term of type T, and let $\mathcal{F}$ be a predicate of type "$\mathsf{Hprop} \to (\llbracket T \rrbracket \to \mathsf{Hprop}) \to \mathsf{Prop}$".*

$$\mathsf{sound}\,\hat{t}\,\mathcal{F} \;\equiv\; \forall H\,Q\,h_i\,h_k.\, \left\{\begin{array}{l} \mathcal{F}\,H\,Q \\ h_i \perp h_k \\ H\;h_i \end{array}\right. \Rightarrow \exists \hat{v}\,h_f\,h_g.\, \left\{\begin{array}{l} h_f \perp h_k \perp h_g \\ \hat{t}_{/h_i+h_k} \Downarrow^{:} \hat{v}_{/h_f+h_k+h_g} \\ Q\;\lceil \hat{v} \rceil\;h_f \end{array}\right.$$

The soundness theorem then asserts that the characteristic formula of a well-typed term is a correct description of the behavior of that term, in the sense that the proposition "$\mathsf{sound}\;\hat{t}\,\llbracket \hat{t} \rrbracket^{\emptyset}$" holds. Before proving the soundness theorem, I first establish a lemma explaining how to eliminate the application of the predicate local that appears at the head of the characteristic formula $\llbracket \hat{t} \rrbracket^{\emptyset}$.

Lemma 7.2.1 (Elimination of local) *Let $\hat{t}$ be a well-typed term of type T, and let $\mathcal{F}$ be a predicate of type "$\mathsf{Hprop} \to (\llbracket T \rrbracket \to \mathsf{Hprop}) \to \mathsf{Prop}$". Then,*

$$\mathsf{sound}\,\hat{t}\,\mathcal{F} \qquad \Rightarrow \qquad \mathsf{sound}\,\hat{t}\,(\mathsf{local}\,\mathcal{F})$$

Proof To prove the lemma, let H and Q be pre- and post-conditions such that "$\mathsf{local}\,\mathcal{F}\,H\,Q$" holds and let h_i and h_k be two disjoint heaps such that $H\;h_i$ holds. The goal is to find the $\hat{v}$, h_f and h_g.

Unfolding the definition of local in the assumption "$\mathsf{local}\,\mathcal{F}\,H\,Q$" and specializing the assumption to the heap h_i, which satisfies H, we obtain some predicates H_i, H_k, H_g and Q_f such that $(H_i * H_k)$ holds of the heap h_i and such that the propositions "$\mathcal{F}\,H_i\,Q_f$" and "$Q_f \star H_k \blacktriangleright Q \star H_g$" hold. By definition of the separating conjunction, the heap h_i can be decomposed into two disjoint heaps h'_i and h'_k such that h'_i satisfies H_i and h'_k satisfies H_k and $h_k = h'_i + h'_k$.

The application of the hypothesis "$\mathsf{sound}\;\hat{t}\,\mathcal{F}$" to the heap h'_i satisfying H_i and to the heap $(h_k + h'_k)$ give the existence of a value $\hat{v}$ and two heaps h'_f and h'_g satisfying the following properties.

$$\hat{t}_{/h'_i+h_k+h'_k} \Downarrow^{:} \hat{v}_{/h'_f+h_k+h'_k+h'_g} \quad \wedge \quad Q_f\;\lceil \hat{v} \rceil\;h'_f$$

The property $Q_f\;\lceil \hat{v} \rceil\;h'_f$ can be exploited with the hypothesis "$Q_f \star H_k \blacktriangleright Q \star H_g$". Since the heap $(h'_f + h'_k)$ satisfies the heap predicate $((Q_f\;\lceil \hat{v} \rceil) * H_k)$, the same heap also satisfies the predicate $((Q\;\lceil \hat{v} \rceil) * H_g)$. By definition of separating conjunction, there exist two heaps h_f and h''_g such that "$Q\;\lceil \hat{v} \rceil\;h_f$" holds and such that the fact $H_g\,h''_g$ and the equality $h'_f + h'_k = h_f + h''_g$ are satisfied.

To conclude, I instantiate h_g as $h'_g + h''_g$. It remains to prove $\hat{t}_{/h_i+h_k} \Downarrow^{:} \hat{v}_{/h_f+h_k+h_g}$. Given the typed reduction sequence obtained earlier on, it suffices to check the equalities $h_i = h'_i + h'_k$ and $h'_f + h'_k + h'_g = h_f + h_g$. □

Theorem 7.2.2 (Soundness) *Let $\hat{t}$ be a typed term.*

$$\vdash \hat{t} \qquad \Rightarrow \qquad \mathit{sound}\ \hat{t}\ [\![\hat{t}]\!]^{\emptyset}$$

Proof The proof goes by induction on the size of the term $\hat{t}$. Here again, all values have size one. Given a typed term $\hat{t}$ of type T, its characteristic formula takes the form $\mathsf{local}\,\mathcal{F}$. I have proved in the previous lemma that, to establish the soundness of $\mathsf{local}\,\mathcal{F}$, it suffices to establish the soundness of $\mathcal{F}$. It therefore remains to study the formula $\mathcal{F}$, whose shape depends on the term $\hat{t}$.

Let H be a pre-condition and Q be a post-condition for $\hat{t}$. For each form that $\hat{t}$ might take, we consider two disjoint heaps h_i and h_k such that the proposition $H\,h_i$ holds. The assumption $\mathcal{F}\,H\,Q$ depends on the shape of $\hat{t}$. The particular form of the assumption is stated in every proof case. The goal is to find a value $\hat{v}$ and two heaps h_f and h_g satisfying $\hat{t}_{/h_i+h_k} \Downarrow^{:} \hat{v}_{/h_f+h_k+h_g}$ and $Q\,\lceil\hat{v}\rceil\,h_f$

• Case "$\hat{t} = \hat{v}$". The assumption is $H \rhd Q\,\lceil\hat{v}\rceil$. Instantiate $\hat{v}$ as $\hat{v}$, h_f as h_i and h_g as the empty heap. It is immediate to check the first conclusion $\hat{v}_{/h_i+h_k} \Downarrow^{:} \hat{v}_{/h_i+h_k}$. The second conclusion, $Q\,\lceil\hat{v}\rceil\,h_i$ is obtained by applying the assumption $H \rhd Q\,\lceil\hat{v}\rceil$ to the assumption "$H\,h_i$".

• Case "$\hat{t} = ((\hat{f}^{\mathsf{func}})\,)(\hat{v}'^{T'})^{T}$". The assumption coming from the characteristic formula is $\mathsf{AppReturns}_{[\![T']\!],[\![T]\!]}\,\lceil\hat{f}\rceil\,\lceil\hat{v}'\rceil\,H\,Q$. Unfolding the definition of $\mathsf{AppReturns}$ and exploiting it on the heaps h_i and h_k gives the existence of a value V of type $[\![T]\!]$ and of two heaps h_f and h_g satisfying the next two properties.

$$\mathsf{AppEval}_{[\![T']\!],[\![T]\!]}\,\lceil\hat{f}\rceil\,\lceil\hat{v}\rceil\,(h_i+h_k)\,V\,(h_f+h_k+h_g) \quad \wedge \quad Q\,V\,h_f$$

Let $\hat{v}$ denote $\lfloor V\rfloor$. The value V is equal to $\lceil\hat{v}\rceil$. So, $Q\,\lceil\hat{v}\rceil\,h_f$ holds. Moreover, by definition of $\mathsf{AppEval}$, the reduction $(\hat{f}\,\hat{v}')_{/(h_i+h_k)} \Downarrow^{:} \hat{v}_{/(h_f+h_k+h_g)}$ holds.

• Case "$\hat{t} = (\mathsf{let}\,x = \hat{t}_1\,\mathsf{in}\,\hat{t}_2)$". The assumption coming from the characteristic formula is

$$\exists Q'.\quad [\![\hat{t}_1]\!]\,H\,Q' \quad \wedge \quad \forall X.\,[\![\hat{t}_2]\!]^{(x\mapsto X)}\,(Q'\,X)\,Q$$

Consider a particular post-condition Q'. The induction hypothesis applied to $\hat{t}_1$ asserts the existence of a value $\hat{v}_1$ and of two heaps h'_f and h'_g satisfying $\hat{t}_{1/h_i+h_k} \Downarrow^{:} \hat{v}_{1/h'_f+h_k+h'_g}$ and $Q\,\lceil\hat{v}_1\rceil\,h'_f$.

I then exploit the second hypothesis by instantiating X as $\lceil\hat{v}_1\rceil$ I obtaining the proposition $[\![\hat{t}_2]\!]^{(x\mapsto\lceil v_1\rceil)}\,(Q'\,\lceil v_1\rceil)\,Q$. By the substitution lemma, this is equivalent to $[\![[x\to v_1]\,\hat{t}_2]\!]^{\emptyset}\,(Q'\,\lceil v_1\rceil)\,Q$. I apply the induction hypothesis on the heap h'_f, which satisfies the precondition $Q'\,\lceil\hat{v}_1\rceil$, and to the heap $h_k + h'_g$, obtaining the existence of a value $\hat{v}$ and of two heaps h_f and h''_g satisfying $\hat{t}_{2/h'_f+h_k+h'_g} \Downarrow^{:} \hat{v}_{/h_f+h_k+h'_g+h''_g}$ and $Q\,\lceil\hat{v}\rceil\,h_f$. Instantiating h_g as $h_k + h_g$ gives the derivation "$\hat{t}_{/h_i+h_k} \Downarrow^{:} \hat{v}_{/h_f+h_k+h_g}$".

• Case "$\hat{t} = (\hat{t}_1\,;\,\hat{t}_2)$". A sequence can be viewed as a let-binding of the form "$\mathsf{let}\,x = \hat{t}_1\,\mathsf{in}\,\hat{t}_2$" for a fresh name x. The soundness of the treatment of let-bindings can therefore be deduced from the previous proof case.

• Case "$\hat{t} = (\mathsf{let}\, x = \hat{w}_1 \,\mathsf{in}\, \hat{t}_2)$". The assumption coming from the characteristic formula is

$$\forall X. \quad X = \lceil \hat{w}_1 \rceil \quad \Rightarrow \quad [\![\hat{t}_2]\!]^{(x \mapsto X)}\, H\, Q$$

I instantiate X with $\lceil \hat{w}_1 \rceil$. The premise is immediately verified and the conclusion gives $[\![\hat{t}_2]\!]^{(x \mapsto \lceil \hat{w}_1 \rceil)}\, H\, Q$. By the substitution lemma, this proposition is equivalent to $[\![[x \to \hat{w}_1]\, \hat{t}_2]\!]^{\emptyset}\, H\, Q$. The conclusion then follows directly from the induction hypothesis.

• Case "$\hat{t} = \mathsf{crash}$". Like in the purely-functional setting.

• Case "$\hat{t} = \mathsf{if}\, \lceil \hat{v} \rceil\, \mathsf{then}\, \hat{t}_1\, \mathsf{else}\, \hat{t}_2$". Similar proof as in the purely-functional setting.

• Case "$\hat{t} = (\mathsf{let\,rec}\, f = \Lambda \overline{A}.\lambda x.\hat{t}_1 \;\mathsf{in}\, \hat{t}_2)$". The assumption of the theorem is "$\forall F.\, \mathcal{H} \Rightarrow [\![\, \hat{t}_2 \,]\!]^{(f \mapsto F)}\, H\, Q$", where $\mathcal{H}$ is

$$\forall \overline{A} X H' Q'.\ [\![\, \hat{t}_1 \,]\!]^{(x \mapsto X, f \mapsto F)}\, H'\, Q' \ \Rightarrow\ \mathsf{AppReturns}\, F\, X\, H'\, Q'.$$

I instantiate the assumption with F as $\lceil \mu f.\Lambda \overline{A}.\lambda x.\hat{t}_1 \rceil$. So, $\mathcal{H}$ implies $[\![\hat{t}_2]\!]^{(f \mapsto F)}\, P$.

The next step consists in proving that $\mathcal{H}$ holds. Consider particular values for $\overline{T}$, X, H' and Q', let $\overline{T}$ denote $\lfloor\!\lfloor \overline{T} \rfloor\!\rfloor$, and assume $[\![\, [\overline{A} \to \overline{T}]\, \hat{t}_1 \,]\!]^{(x \mapsto X, f \mapsto F)}\, H'\, Q'$. The goal is $\mathsf{AppReturns}\, F\, X\, H'\, Q'$. Let $\hat{v}_1$ denote $\lfloor X \rfloor$. So, $\lceil \hat{v}_1 \rceil$ is equal to X. By the substitution lemma, the assumption $[\![\, \hat{t}_1 \,]\!]^{(x \mapsto X, f \mapsto F)}\, H'\, Q'$ is equivalent to $[\![\, [f \to \mu f.\Lambda \overline{A}.\lambda x.\hat{t}_1]\, [x \to \hat{v}_1]\, [\overline{A} \to \overline{T}]\, \hat{t}_1 \,]\!]^{\emptyset}\, H'\, Q'$. Thereafter, let $\hat{t}'$ denote the term $[f \to \mu f.\Lambda \overline{A}.\lambda x.\hat{t}_1]\, [x \to \hat{v}_1]\, [\overline{A} \to \overline{T}]\, \hat{t}_1$. The assumption becomes $[\![\hat{t}']\!]^{\emptyset}\, H'\, Q'$.

By definition of $\mathsf{AppReturns}$, the goal $\mathsf{AppReturns}\, F\, X\, H'\, Q'$ is equivalent to:

$$\begin{array}{l} \forall h'_i\, h'_k.\ (h'_i \perp h'_k) \ \wedge\ H'\, h'_i \ \Rightarrow \\ \quad \exists V'\, h'_f\, h'_g.\ \mathsf{AppEval}\, F\, X\, (h'_i + h'_k)\, V'\, (h'_f + h'_k + h'_g) \ \wedge\ Q'\, V'\, h'_f \end{array}$$

Given particular heaps h'_i and h'_k, I invoke the induction hypothesis on them and on the hypothesis $[\![\hat{t}']\!]^{\emptyset}\, H'\, Q'$. This gives the existence of a value $\hat{v}'$ of type T' and of two heaps h'_f and h'_g satisfying $\hat{t}'_{/h'_i + h'_k} \Downarrow^{:} \hat{v}'_{/h'_f + h'_k + h'_g}$ and $Q'\, \lceil \hat{v}' \rceil\, h'_f$. I then instantiate V' as $\lceil v' \rceil$. The proof obligation $Q'\, V'\, h'_f$ is immediately solved. There remains to establish the following fact.

$$\mathsf{AppEval}\, \lceil \mu f.\Lambda \overline{A}.\lambda x.\hat{t}_1 \rceil\, \lceil \hat{v}_1 \rceil\, (h'_i + h'_k)\, \lceil \hat{v}' \rceil\, (h'_f + h'_k + h'_g)$$

By definition of $\mathsf{AppEval}$, this fact is "$((\mu f.\Lambda \overline{A}.\lambda x.\hat{t}_1)\, \hat{v}_1)_{/h'_i + h'_k} \Downarrow^{:} \hat{v}'_{/h'_f + h'_k + h'_g}$". The reduction holds because contraction of the redex "$((\mu f.\Lambda \overline{A}.\lambda x.\hat{t}_1)\, \hat{v}_1)$" is equal "$(\mu f.\Lambda \overline{A}.\lambda x.\hat{t}_1)\, \hat{v}_1$", which reduces towards t'.

Now that we have proved the premise $\mathcal{H}$, we can conclude using the assumption $[\![\, \hat{t}_2 \,]\!]^{(f \mapsto F)}\, H\, Q$. By the substitution lemma, we get $[\![\, [f \to \mu f.\Lambda \overline{A}.\lambda x.\hat{t}_1]\, \hat{t}_2 \,]\!]^{\emptyset}\, H\, Q$. The conclusion follow from the induction hypothesis applied to that fact.

• Case "$\hat{t} = (\mathsf{let\,rec}\, f = \Lambda \overline{A}.\lambda x y.\hat{t}_1 \;\mathsf{in}\, \hat{t}_2)$". As explained earlier on (§7.1.4), I have to prove the soundness of the body description generated for functions of two arguments. Let $\hat{f}$ denote the function closure "$\mu f.\Lambda \overline{A}.\lambda x y.\hat{t}_1$", which is the same

as "$\mu f.\Lambda\overline{A}.\lambda x.(\mathsf{let\,rec}\, g = \lambda y.\, \hat{t}_1\, \mathsf{in}\, g)$". The body description of $\hat{f}$ is equivalent to "$\forall \overline{A} X P.\ (\forall G.\ \mathcal{H} \Rightarrow P\, G) \Rightarrow \mathsf{AppPure}\lceil \hat{f} \rceil X P$", where

$$\mathcal{H} \quad \equiv \quad (\forall Y H' Q'.\ [\![\hat{t}_1]\!]^{(x \mapsto X, y \mapsto Y)}\, H'\, Q' \ \Rightarrow\ \mathsf{AppReturns}_1\, G\, Y\, H'\, Q')$$

Let $\overline{\mathcal{T}}$ be somes types, and let $\overline{T}$ denote $\lfloor\!\lfloor \overline{\mathcal{T}} \rfloor\!\rfloor$. Let X be a given argument and let P be a given predicate of type $\mathsf{Func} \to \mathsf{Prop}$ such that "$\forall G.\ \mathcal{H} \Rightarrow P\, G$". The goal is to prove $\mathsf{AppPure}\lceil \hat{f} \rceil X P$. By definition of $\mathsf{AppPure}$, the goal unfolds to "$\exists V.\ (\forall h.\, \mathsf{AppEval}\, \lceil \hat{f} \rceil\, X\, h\, V\, h) \wedge P\, V$". In what follows, let $\hat{g}$ denote the function "$\mu g.\Lambda.\lambda y.[x \to \hat{v}]\, [\overline{A} \to \overline{T}]\, \hat{t}_1$". I instantiate V as $\lceil \hat{g} \rceil$.

Let h be a heap and $\hat{m}$ denote $\lfloor h \rfloor$. Let $\hat{v}$ denote $\lfloor X \rfloor$. So, X is equal to $\lceil \hat{v}_2 \rceil$. The proposition "$\mathsf{AppEval}\, \lceil \hat{f} \rceil\, X\, h\, V\, h$" is equivalent to "$\mathsf{AppEval}\, \lceil \hat{f} \rceil\, \lceil \hat{v}_2 \rceil\, \lceil \hat{m} \rceil\, \lceil \hat{g} \rceil\, \lceil \hat{m} \rceil$", and it follows from the typed reduction sequence:

$$((\mu f.\Lambda\overline{A}.\lambda x.(\mathsf{let\,rec}\, g = \Lambda.\lambda y.\hat{t}_1\ \mathsf{in}\ g))\, \hat{v}_2)_{/\hat{m}} \Downarrow^{:} (\mu g.\Lambda.\lambda y.[x \to \hat{v}_2]\, [\overline{A} \to \overline{T}]\, \hat{t}_1)_{/\hat{m}}$$

It remains to prove $P\, \lceil \hat{g} \rceil$. To that end, I apply the assumption "$\forall G.\ \mathcal{H} \Rightarrow P\, G$". The goal to be established is "$[G \to \lceil \hat{g} \rceil]\, \mathcal{H}$". Let $\hat{t}'_1$ stand for the term $[x \to \hat{v}]\, [\overline{A} \to \overline{T}]\, \hat{t}_1$. Using the substitution lemma, the goal can be rewritten as:

$$\forall Y H' Q'.\ [\![\, \hat{t}'_1 \,]\!]^{(y \mapsto Y)}\, H'\, Q' \ \Rightarrow\ \mathsf{AppReturns}_1\, \lceil \mu g.\Lambda.\lambda y.\hat{t}'_1 \rceil\, Y\, H'\, Q'$$

This proposition is exactly the body description of the unary function $\hat{g}$. The soundness of body description for unary functions has already been established in the previous proof case.

• Case "$\hat{t} = (\mathsf{while}\, \hat{t}_1\, \mathsf{do}\, \hat{t}_2)$". The assumption given by the characteristic formula is the proposition "$\forall R.\ \mathsf{is\_local}\, R \wedge \mathcal{H} \Rightarrow R\, H\, Q$", where

$$\mathcal{H} \quad \equiv \quad \forall H' Q'.\ [\![\mathsf{if}\, t_1\, \mathsf{then}\, (t_2\, ;\, |R|)\, \mathsf{else}\, tt]\!]\, H'\, Q' \ \Rightarrow\ R\, H'\, Q'$$

We know that h_i be a heap satisfying H, and the goal is to find two heaps h_f and h_g such that $\hat{t}_{/h_i+h_k} \Downarrow^{:} tt_{/h_f+h_k+h_g}$ and $Q\ tt\ h_f$.

The soundness of a while-loop relies on the soundness of its encoding as a recursive function, called $\hat{t}'$. It is straightforward to check that the term $\hat{t}'$, shown next, has exactly the same semantics as $\hat{t}$.

$$\hat{t}' \quad \equiv \quad (\mathsf{let\,rec}\, f = \lambda\_.\, (\mathsf{if}\, \hat{t}_1\, \mathsf{then}\, (\hat{t}_2\, ;\, f\ tt)\, \mathsf{else}\, tt)\, \mathsf{in}\, f\ tt)$$

So, the goal reformulates as $\hat{t}'_{/h_i+h_k} \Downarrow^{:} tt_{/h_f+h_k+h_g}$ and $Q\ tt\ h_f$. The induction hypothesis applied to $\hat{t}'$ asserts that the proposition $[\![\, \hat{t}' \,]\!]^{\emptyset}\, H\, Q$ is a sufficient condition for proving this goal. It thus remains to establish that the characteristic formula for $\hat{t}$ is stronger than the characteristic formula for $\hat{t}'$.

A partial computation of the formula $[\![\, \hat{t}' \,]\!]^{\emptyset}\, H\, Q$ gives the proposition "$\forall f.\ \mathcal{H}' \Rightarrow \mathsf{AppReturns}\, f\ tt\, H\, Q$", where

$$\mathcal{H}' \quad \equiv \quad \forall H' Q'.\ [\![\mathsf{if}\, t_1\, \mathsf{then}\, (t_2\, ;\, f\ tt)\, \mathsf{else}\, tt]\!]^{\emptyset}\, H'\, Q' \ \Rightarrow\ \mathsf{AppReturns}\, f\ tt\, H'\, Q'$$

Let f be a Coq value of type func satisfying $\mathcal{H}'$. To prove $\mathsf{AppReturns}\ f\ tt\ H\ Q$, I apply the assumption "$\forall R.\ \mathsf{is\_local}\ R \wedge \mathcal{H} \Rightarrow R\,H\,Q$" with R instantiated as $\mathsf{AppReturns} f\,tt$. Note that R is then a local predicate, as required. There remains to prove the proposition $[R \rightarrow \mathsf{AppReturns} f\,tt]\,\mathcal{H}$. This proposition is exactly the assumption $\mathcal{H}'$, since $[\![\,|\mathsf{AppReturns} f\,tt|\,]\!]$ is the same as $\mathsf{AppReturns} f\,tt$. After all, the fact that $[R \rightarrow \mathsf{AppReturns} f\,tt]\,\mathcal{H}$ matches the assumption $\mathcal{H}'$ follows from the design of the characteristic formula of a while loop as (a simplified version of) the characteristic formula of its recursive encoding.

• Case "$\hat{t} = (\mathsf{for}\ x = n\ \mathsf{to}\ n'\ \mathsf{do}\ \hat{t}_1)$". The proof of soundness for for-loops is very similar to that given for while-loops, so I only give the main steps. The characteristic formula for $\hat{t}$ is "$\forall S.\ \mathsf{is\_local}_1\ S \wedge \mathcal{H} \Rightarrow S\,a\,H\,Q$", where

$$\mathcal{H} \quad \equiv \quad \forall i H' Q'.\ [\![\mathsf{if}\ i \le b\ \mathsf{then}\ (t\,;\ |S\,(i+1)|)\ \mathsf{else}\ tt]\!]\,H'\,Q' \Rightarrow S\,i\,H'\,Q'$$

The term $\hat{t}'$ that corresponds to the encoding of $\hat{t}$ is defined as follows.

$$\hat{t}' \quad \equiv \quad (\mathsf{let\ rec}\ f = \lambda i.\,(\mathsf{if}\ i \le b\ \mathsf{then}\ (\hat{t}_1\,;\ f\,(i+1))\ \mathsf{else}\ tt)\ \mathsf{in}\ f\,a)$$

The characteristic formula of $\hat{t}'$ is "$\forall f.\ \mathcal{H}' \Rightarrow \mathsf{AppReturns}\ f\,a\,H\,Q$", where

$$\mathcal{H}' \ \equiv \ \forall i H' Q'.\ [\![\mathsf{if}\ i \le b\ \mathsf{then}\ (t_1\,;\ f\,(i+1))\ \mathsf{else}\ tt]\!]^{\emptyset}\,H'\,Q' \Rightarrow \mathsf{AppReturns}\ f\,i\,H'\,Q'$$

To prove the characteristic formula of $\hat{t}'$ using that of $\hat{t}$, it suffices to prove that $\mathcal{H}'$ implies $\mathcal{H}$, which is obtained by instantiating S as $\mathsf{AppReturns}\,f$.

There remains to prove the soundness of the specification of the primitive functions.

• Case ref. The specification is "$\mathsf{Spec}\,\mathsf{ref}\,(\lambda V\,R.\ R\,[\,]\,(\lambda L.\,L \hookrightarrow_{\mathcal{T}} V))$", where $\mathcal{T}$ is the type of the argument V. Considering the definition of Spec, this is equivalent to showing that the relation "$\mathsf{AppReturns}_{\mathcal{T},\mathsf{Loc}}\,\mathsf{ref}\,V\,[\,]\,(\lambda L.\,L \hookrightarrow_{\mathcal{T}} V)$" holds for any value V of type $\mathcal{T}$. By definition of $\mathsf{AppReturns}$, the goal is to show

$$\begin{array}{l}\forall h_i\,h_k.\ (h_i \perp h_k) \ \wedge \ [\,]\,h_i \ \Rightarrow \ \exists L\,h_f\,h_g. \\ \qquad \mathsf{AppEval}_{\mathcal{T},\mathsf{Loc}}\,\mathsf{ref}\,V\,(h_i+h_k)\,L\,(h_f+h_k+h_g) \ \wedge \ (L \hookrightarrow_{\mathcal{T}} V)\,h_f\end{array}$$

The hypothesis $[\,]\,h_i$ asserts that h_i is empty, meaning that the evaluation of ref takes place in a heap h_k. Let $\hat{v}$ be equal to $\lfloor V \rfloor$. So, V is equal to $\lceil \hat{v} \rceil$. The reduction rule for ref asserts that $(\mathsf{ref}\,\hat{v})^{\mathsf{Loc}}{}_{/h_k} \Downarrow l^{\mathsf{loc}}{}_{/h_k \uplus [l \mapsto \lceil \hat{v} \rceil]}$ holds for some location l fresh from the domain of h_k. To conclude, it suffices to instantiate L as $\lceil l \rceil$, h_f as the singleton heap $(L \rightarrow_{\mathcal{T}} V)$ and h_g as the empty heap.

• Case get. "$\mathsf{Spec}\,\mathsf{get}\,(\lambda L\,R.\ \forall V.\ R\,(L \hookrightarrow_{\mathcal{T}} V)\,(\backslash{=}\,V \star (L \hookrightarrow_{\mathcal{T}} V)))$" is the specification, where $\mathcal{T}$ is the type of V. Considering the definition of Spec, it suffices to prove that "$\mathsf{AppReturns}_{\mathsf{Loc},\mathcal{T}}\,\mathsf{get}\,L\,(L \hookrightarrow_{\mathcal{T}} V)\,(\backslash{=}\,V \star (L \hookrightarrow_{\mathcal{T}} V))$" holds for any location L and any value V. Let L be a particular location and V be a particular

value of type $\mathcal{T}$. By definition of AppReturns, the goal is:

$$\forall h_i\, h_k.\ (h_i \perp h_k)\ \wedge\ (L \hookrightarrow_{\mathcal{T}} V)\, h_i\ \Rightarrow\ \exists V'\, h_f\, h_g.\ \begin{cases} \mathsf{AppEval}_{\mathsf{Loc},\mathcal{T}}\ \mathsf{get}\, L\, (h_i + h_k)\, V'\, (h_f + h_k + h_g) \\ V' = V \\ (L \hookrightarrow_{\mathcal{T}} V)\, h_f \end{cases}$$

To prove the goal, I instantiate V' as V, h_f as h_i and h_g as the empty heap. There remains to prove $\mathsf{AppEval}\, \mathsf{get}\, L\, (h_i + h_k)\, V\, (h_i + h_k)$. The hypothesis $(L \hookrightarrow_{\mathcal{T}} V)\, h_i$ asserts that h_i is a singleton heap of the form $(l \rightarrow_{\mathcal{T}} V)$. Let l denote the location $\lfloor L \rfloor$ and let $\hat{v}$ denote the value $\lfloor V \rfloor$. The value $\hat{v}$ has type $[\![\mathcal{T}]\!]$, which is written T in what follows. The goal can be reformulated as the following statement: $\mathsf{AppEval}_{\mathsf{Loc},[\![T]\!]}\, \mathsf{get}\, \lceil l \rceil\, ((l \rightarrow V) + h_k)\, \lceil \hat{v} \rceil\, ((l \rightarrow V) + h_k)$. By definition of AppEval, it suffices to prove $(\mathsf{get}\, l)^{T}{}_{/(l \rightarrow \lceil \hat{v} \rceil) + h_k} \Downarrow \hat{v}^{T}{}_{/(l \rightarrow \lceil \hat{v} \rceil) + h_k}$. This proposition follows directly from the reduction rule for get.

• Case set. "$\mathsf{Spec}\, \mathsf{set}\, (\lambda(L, V)\, R.\ \forall V'.\ R\, (L \hookrightarrow_{\mathcal{T}'} V')\, (\#\, (L \hookrightarrow_{\mathcal{T}} V)))$" is the specification. The goal is to prove that "$\mathsf{AppReturns}_{(\mathsf{Loc} \times \mathcal{T}),\mathsf{Unit}}\, \mathsf{get}\, (L, V)\, (L \hookrightarrow_{\mathcal{T}'} V')\, (\#\, (L \hookrightarrow_{\mathcal{T}} V))$" holds for any location L and for any values V and V'. Given such parameters, the goal is equivalent to:

$$\forall h_i\, h_k.\ (h_i \perp h_k)\ \wedge\ (L \hookrightarrow_{\mathcal{T}'} V')\, h_i\ \Rightarrow\ \exists V''\, h_f\, h_g.\ \begin{cases} \mathsf{AppEval}_{(\mathsf{Loc} \times \mathcal{T}),\mathsf{unit}}\ \mathsf{set}\, (L, V)\, (h_i + h_k)\, V''\, (h_f + h_k + h_g) \\ V'' = tt \\ (L \hookrightarrow_{\mathcal{T}} V)\, h_f \end{cases}$$

I instantiate V'' as $\lceil tt \rceil$, h_f as $(L \rightarrow_T V)$ and h_g as the empty heap. Let l denote $\lfloor L \rfloor$, $\hat{v}$ denote $\lfloor V \rfloor$ and $\hat{v}'$ denote $\lfloor V' \rfloor$. The assumption $(L \hookrightarrow_{\mathcal{T}'} V')\, h_i$ asserts that h_i is of the form $(l \rightarrow_{\mathcal{T}'} \hat{v}')$. There remains to prove the following proposition: $\mathsf{AppEval}\, \mathsf{set}\, \lceil (l, \hat{v}) \rceil\, ((l \rightarrow \hat{v}') + h_k)\, \lceil tt \rceil\, ((l \rightarrow \hat{v}) + h_k)$. This proposition follows from the reduction rule for set, since $(\mathsf{set}\, l\, \hat{v})^{\mathsf{unit}}{}_{/(l \rightarrow \hat{v}') + h_k} \Downarrow tt^{\mathsf{unit}}{}_{/(l \rightarrow \hat{v}) + h_k}$.

• Case cmp. The proof is straightforward since cmp does not involve any reasoning on the heap.

7.3 Completeness

The proof of completeness of characteristic formulae for imperative programs involves two additional ingredients compared with the corresponding proof from the purely functional setting. First, it involves the notion of most-general heap specification, which asserts that, for every value stored in the heap, the most-general specification of that value can be assumed to hold. Second, the definition of most-general specifications needs to take into account the fact that the exact memory address of a location cannot be deduced from characteristic formulae. Indeed, the specification of the allocation function ref simply describes the creation of one fresh location, but without revealing its address. As a consequence, the heap can only be

specified up to renaming. The practical implication is that all the predicates become parameterized by a finite map from program locations to program locations, written α, whose purpose is to rename all the locations occurring in values, terms and heaps.

7.3.1 Most-general specifications

The notion of labelling of terms and the definition of correct labelling with respect to a set E of values of type Func are the same as in the previous chapter. The definition of body description is also the the same as before, except that it includes a pre-condition.

Definition 7.3.1 (Body description of a labelled function)

$$\begin{array}{l} \mathit{body}\ (\mu f.\Lambda\overline{A}.\lambda x.\tilde{t})^{\{F\}} \ \equiv \\ \quad \forall\overline{A}.\ \forall X.\forall H.\forall Q.\ (\llbracket\, \tilde{t}\, \rrbracket^{(x\mapsto X, f\mapsto F)}\ H\ Q) \ \Rightarrow\ \mathit{AppReturns}\, F\ X\ H\ Q \end{array}$$

The most-general specification predicate from the previous chapter is recalled next. The predicate $\mathsf{mgs}\,\hat{v}$ holds of a value V if there exists a correct labelling $\tilde{v}$ of the value $\hat{v}$ with constants from some set E, such that V is equal to the decoding of $\tilde{v}$.

Definition 7.3.2 (Most-general specification of a value, old version) *Let $\hat{v}$ be a well-typed value.*

$$\mathit{mgs}\,\hat{v} \quad \equiv \quad \lambda V.\ \exists E.\ \exists\tilde{v}.\ \hat{v} = \mathit{strip\_labels}\,\tilde{v}\ \wedge\ \mathit{labels}\,E\,\tilde{v}\ \wedge\ V = \lceil\tilde{v}\rceil$$

The definition used in this chapter extends it with a renaming map for locations, that is, a finite map binding locations to locations. Let $\mathsf{locs}(\hat{v})$ denote the set of locations occurring in the value $\hat{v}$. Let "$\alpha\,\hat{v}$" denote the renaming of the locations occurring in $\hat{v}$ according to the map α. The new specification predicate, written $\mathsf{mgv}\,\alpha\,\hat{v}$, corresponds to the most-general specification of the value $(\alpha\,\hat{v})$.

Definition 7.3.3 (Most-general specification of a value) *Let $\hat{v}$ be a well-typed value, and α be a map from locations to locations.*

$$\mathit{mgv}\,\alpha\,\hat{v} \quad \equiv \quad \lambda V.\quad \mathit{mgs}\,(\alpha\,\hat{v})\,V \quad \wedge \quad \mathit{locs}(\hat{v}) \subseteq \mathrm{dom}(\alpha)$$

The definition of mgv also includes a side-condition to ensure that the domain of α covers all the locations occurring in the value $\hat{v}$. The role of this side-condition is to guarantee that the meaning of the predicate $\mathsf{mgv}\,\alpha\,\hat{v}$ is preserved through an extension of the map α. Indeed, when all the locations of $\hat{v}$ are covered by α, then for any α' extends α, the value $(\alpha'\,\hat{v})$ is equal to the value $(\alpha\,\hat{v})$.

The expression "$\alpha\,\hat{m}$" denotes the application of the renaming α both to the domain of the map $\hat{m}$ and to the values in the range of $\hat{m}$. The predicate "$\mathsf{mgh}\,\alpha\,\hat{m}$" describes the most-general specification of a typed store $\hat{m}$ with respect to a renaming map α. If h is a heap, then the proposition "$\mathsf{mgh}\,\alpha\,\hat{m}\,h$" holds at the following two conditions. First, the domain of h should correspond to the renaming of the

domain of $\hat{m}$ with respect to α. Second, for every value $\hat{v}$ stored at the location l in $\hat{m}$, the value V stored at the corresponding location $\alpha\, l$ in h should satisfy the most-general specification of $\hat{v}$ with respect to α. Technically, the proposition "$\mathsf{mgv}\,\alpha\,\hat{v}V$" should hold. In the definition of mgh shown next, the value $\hat{v}$ is being referred to as $\hat{m}[l]$, and the value $h[\alpha\, l]$ contained in the heap h is a dependent pair made of the type $\mathcal{T}$ and of the value V of type $\mathcal{T}$.

Definition 7.3.4 (Most-general specification of a store) *Let m be a well-typed store and let α be a renaming map for locations.*

$$\mathit{mgh}\,\alpha\,\hat{m} \quad \equiv \quad \lambda h.\ \begin{cases} \mathrm{dom}(h) = \mathrm{dom}(\alpha\,\hat{m}) \\ \forall l \in \mathrm{dom}(\hat{m}).\ \mathit{mgv}\,\alpha\,(\hat{m}[l])\,V \\ \qquad\qquad \textit{where}\ (\mathcal{T}, V) = h[\alpha\, l] \\ \mathit{locs}(\hat{m}) \subseteq \mathrm{dom}(\alpha) \end{cases}$$

Again, a side-condition is involved to assert that all the locations occurring in the domain or in the range of $\hat{m}$ are covered by the renaming α.

It remains to define the notion of most-general post-condition of a term. The proposition "$\mathsf{mgp}\,\alpha\,\hat{v}\,\hat{m}$" describes the most-general post-condition that can be assigned to a term whose evaluation returns the typed value $\hat{v}$ in a typed memory state $\hat{m}$. The predicate $\mathsf{mgp}\,\alpha\,\hat{v}\,\hat{m}$ holds of a value V and of the heap h if there exists a map α' that extends α such that the value V satisfies the most-general specification of the value $\hat{v}$ modulo α' and such that the heap h satisfies the most-general specification of the store $\hat{m}$ modulo α'.

Definition 7.3.5 (Most-general post-condition) *Let $\hat{v}$ be a well-typed value, let $\hat{m}$ be a well-typed store, and let α be a renaming map for locations.*

$$\mathit{mgp}\,\alpha\,\hat{v}\,\hat{m} \quad \equiv \quad \lambda V.\ \lambda h.\ \exists \alpha'.\ \alpha' \sqsupseteq \alpha \ \wedge\ \mathit{mgv}\,\alpha'\,\hat{v}\,V \ \wedge\ \mathit{mgh}\,\alpha'\,\hat{m}\,h$$

Remark: the predicate "$\mathsf{mgp}\,\alpha\,\hat{v}\,\hat{m}$" can also be formulated with heap predicates, as shown next.

$$\mathsf{mgp}\,\alpha\,\hat{v}\,\hat{m} \quad = \quad \lambda V.\ \exists \alpha'.\ [\alpha' \sqsupseteq \alpha] * [\mathsf{mgv}\,\alpha'\,\hat{v}\,V] * (\mathsf{mgh}\,\alpha'\,\hat{m})$$

The renaming map α is extended every time a new location is being allocated. The following lemma explains that the predicates mgv and mgh are covariant in the argument α and that the predicate mgp is contravariant in its argument α.

Lemma 7.3.1 (Preservation through extension of the renaming) *Let α and α' be two renamings for locations.*

$$\begin{array}{lcll} \alpha' \sqsupseteq \alpha & \wedge & \mathit{mgv}\,\alpha\,\hat{v}\,V & \Rightarrow\ \mathit{mgv}\,\alpha'\,\hat{v}\,V \\ \alpha' \sqsupseteq \alpha & \wedge & \mathit{mgh}\,\alpha\,\hat{m}\,h & \Rightarrow\ \mathit{mgh}\,\alpha'\,\hat{m}\,h \\ \alpha \sqsupseteq \alpha' & \wedge & \mathit{mgp}\,\alpha\,\hat{v}\,\hat{m}\,V\,h & \Rightarrow\ \mathit{mgp}\,\alpha'\,\hat{v}\,\hat{m}\,V\,h \end{array}$$

Proof For mgv, the assumption $\mathsf{locs}(\hat{v}) \subseteq \mathrm{dom}(\alpha)$ implies that $(\alpha'\,\hat{v})$ is equal to $(\alpha\,\hat{v})$. For mgh, the assumption $\mathsf{locs}(\hat{m}) \subseteq \mathrm{dom}(\alpha)$ implies that $(\alpha'\,\hat{m})$ is equal to $(\alpha\,\hat{m})$ and that for any location l in the domain of m, $(\alpha'\,l)$ is equal to $(\alpha\,l)$. For mgp, the hypothesis $\mathsf{mgp}\,\alpha\,\hat{v}\,\hat{m}\,V\,h$ asserts the existence of some map α'' such that $\alpha'' \sqsupseteq \alpha$ and $\mathsf{mgv}\,\alpha''\,\hat{v}\,V$ and $\mathsf{mgh}\,\alpha''\,\hat{m}\,h$. To prove the conclusion $\mathsf{mgp}\,\alpha'\,\hat{v}\,\hat{m}\,V\,h$, it suffices to observe that the map α'' extends the map α and that the map α extends the map α', so by transitivity α'' extends α'. □

7.3.2 Completeness theorem

Theorem 7.3.1 (Completeness for full, well-typed executions) *Let $\hat{t}$ be a closed term that does not contain any location nor any function closure, let $\hat{v}$ be a typed value, let $\hat{m}$ be a typed store.*

$$\vdash \hat{t} \quad \wedge \quad \hat{t}_{/\emptyset} \Downarrow^{:} \hat{v}_{/\hat{m}} \quad \Rightarrow \quad [\![\,\hat{t}\,]\!]^{\emptyset}\,[\,]\,(\lambda V.\,\exists\alpha.\,[\mathit{mgv}\,\alpha\,\hat{v}\,V] * (\mathit{mgh}\,\alpha\,\hat{m}))$$

Proof I prove by induction on the derivation of the reduction judgment that if a term $\hat{t}$ in a store $\hat{m}$ reduces to a value $\hat{v}$ in a store $\hat{m}'$, then for any correct labelling $\tilde{t}$ of $\hat{t}$ and for any renaming of locations α, the characteristic formula of $(\alpha\,\tilde{t})$ holds of the pre-condition that corresponds to the most-general specification of the store $\hat{m}$ and of the post-condition that corresponds to the most-general specification associated with the value $\hat{v}$ and the store $\hat{m}'$. The formal statement, which follows, includes side-conditions asserting that the domain of the renaming map for locations α should correspond exactly to the domain of the store $\hat{m}$, and that the locations occurring in the term $\hat{t}$ are actually allocated in the sense that they belong to the domain of the store $\hat{m}$.

$$\hat{t}_{/\hat{m}} \Downarrow^{:} \hat{v}_{/\hat{m}'} \;\Rightarrow\; \left\{\begin{array}{l} \forall\tilde{t}\,\alpha\,E. \\ \quad \vdash \hat{t} \\ \quad \vdash \hat{m} \\ \quad \hat{t} = \mathsf{strip\_labels}\,\tilde{t} \\ \quad \mathsf{labels}\,E\,(\alpha\,\tilde{t}) \\ \quad \mathrm{dom}(\alpha) = \mathrm{dom}(\hat{m}) \\ \quad \mathsf{locs}(\hat{t}) \subseteq \mathrm{dom}(\hat{m}) \end{array}\right. \Rightarrow\; [\![\,\alpha\,\tilde{t}\,]\!]^{\emptyset}\,(\mathsf{mgh}\,\alpha\,\hat{m})\,(\mathsf{mgp}\,\alpha\,\hat{v}\,\hat{m}')$$

There is one case per possible reduction rule. In each case, the characteristic formula starts with the predicate local, which I directly eliminate (recall that $\mathcal{F}\,H\,Q$ always implies $\mathsf{local}\,\mathcal{F}\,H\,Q$). Henceforth, I refer to the pre-condition "$\mathsf{mgh}\,\alpha\,\hat{m}$" as H and to the post-condition "$\mathsf{mgp}\,\alpha\,\hat{v}\,\hat{m}'$" as Q.

• Case $\hat{v}_{/\hat{m}} \Downarrow^{:} \hat{v}_{/\hat{m}}$. The term $\tilde{t}$ labels the term $\hat{v}$. Let $\tilde{v}$ denote the labelling of the value $\hat{v}$. The goal is to prove $H \rhd Q\,\lceil\alpha\,\tilde{v}\rceil$, which unfolds to:

$$\forall h.\quad \mathsf{mgh}\,\alpha\,\hat{m}\,h \quad \Rightarrow \quad \exists\alpha'.\ \alpha' \sqsupseteq \alpha \ \wedge\ \mathsf{mgv}\,\alpha'\,\hat{v}\,\lceil\alpha\,\tilde{v}\rceil \ \wedge\ \mathsf{mgh}\,\alpha'\,\hat{m}\,h$$

The conclusion is obtained by instantiating α' as α. The proposition $\mathsf{mgh}\,\alpha\,\hat{m}\,h$ holds by assumption, and the proposition "$\mathsf{mgv}\,\alpha\,\hat{v}\,\lceil\alpha\,\tilde{v}\rceil$" holds by definition of mgv, using the fact that "$\mathsf{labels}\,E\,(\alpha\,\tilde{v})$" holds.

• Case $(\mathsf{let}\, x = \hat{t}_1 \,\mathsf{in}\, \hat{t}_2)_{/\hat{m}} \Downarrow^{:} \hat{v}_{/\hat{m}'}$ with $\hat{t}_{1/\hat{m}} \Downarrow^{:} \hat{v}_{1/m''}$ and $([x \to \hat{v}_1]\, \hat{t}_2)_{/m''} \Downarrow^{:} \hat{v}_{/\hat{m}'}$. Let $\tilde{t}_1$ and $\tilde{t}_2$ be the labelled subterms associated with $\hat{t}_1$ and $\hat{t}_2$, respectively. Let T_1 denote the type of x. The goal is:

$$\exists Q'.\quad [\![\, \alpha\, \tilde{t}_1 \,]\!]^{\emptyset}\, H\, Q' \quad\wedge\quad \forall X.\ [\![\, \alpha\, \tilde{t}_2 \,]\!]^{(x \mapsto X)}\, (Q'\, X)\, Q$$

To prove this goal, I instantiate Q' as the predicate $\mathsf{mgp}\, \alpha\, \hat{v}_1\, m''$. The first subgoal is "$[\![\, \alpha\, \tilde{t}_1 \,]\!]^{\emptyset}\, H\, (\mathsf{mgp}\, \alpha\, \hat{v}_1\, m'')$". It follows from the induction hypothesis.

For the second subgoal, let X be a value of type $[\![T_1]\!]$. We need to establish the proposition "$[\![\, \alpha\, \tilde{t}_2 \,]\!]^{(x \mapsto X)}\, (Q'\, X)\, Q$". By unfolding the definition of Q' and applying the extraction rules for local predicates, the goal becomes:

$$\forall \alpha''.\ \alpha'' \sqsupseteq \alpha \ \wedge\ \mathsf{mgv}\, \alpha''\, \hat{v}_1\, X \ \Rightarrow\ [\![\, \alpha\, \tilde{t}_2 \,]\!]^{(x \mapsto X)}\, (\mathsf{mgh}\, \alpha''\, m'')\, Q$$

Let α'' be a renaming map that extends α. By definition of mgv, there exists a set E' and a labelling $\tilde{v}_1$ of $\hat{v}_1$ such that "$\mathsf{labels}\ E'\ (\alpha''\, \tilde{v}_1)$" and such that X is equal to $\lceil \alpha''\, \tilde{v}_1 \rceil$, moreover satisfying the inclusion "$\mathsf{locs}(\hat{v}_1) \subseteq \mathrm{dom}(\alpha'')$". From the pre-condition $(\mathsf{mgh}\, \alpha''\, m'')$, we can extract the hypothesis that $\mathrm{dom}(\alpha'')$ is equal to $\mathrm{dom}(m'')$. This fact is used later on.

By the substitution lemma, the formula "$[\![\, \alpha\, \tilde{t}_2 \,]\!]^{(x \mapsto X)}$" is equivalent to "$[\![\, [x \to \alpha''\, \tilde{v}_1]\, (\alpha\, \tilde{t}_2) \,]\!]^{\emptyset}$". Since $\mathsf{locs}(\hat{t}_2) \sqsubseteq \mathrm{dom}(\alpha)$ and $\alpha'' \sqsupseteq \alpha$, the $(\alpha\, \tilde{t}_2)$ is equal to $(\alpha''\, \tilde{t}_2)$. So, it remains to prove the following proposition.

$$[\![\, \alpha''\, ([x \to \tilde{v}_1]\, \tilde{t}_2) \,]\!]^{\emptyset}\, (\mathsf{mgh}\, \alpha''\, m'')\, (\mathsf{mgp}\, \alpha\, \hat{v}\, \hat{m}')$$

Since mgp is contravariant in the renaming map, and since the map α'' extends the map α, the post-condition "$\mathsf{mgp}\, \alpha\, \hat{v}\, \hat{m}'$" can be strengthened by the rule of consequence into the post-condition "$\mathsf{mgp}\, \alpha''\, \hat{v}\, \hat{m}'$".

The strengthened goal then follows from the induction hypothesis. The premises can be checked as follows. First, the set $E \cup E'$ correctly labels the term $(\alpha''\, ([x \to \tilde{v}_1]\, \tilde{t}_2))$ because both "$\mathsf{labels}\ E'\ (\alpha''\, \tilde{v}_1)$" and "$\mathsf{labels}\ E\ (\alpha\, \tilde{t}_2)$" are true and because $(\alpha\, \tilde{t}_2)$ is equal to $(\alpha''\, \tilde{t}_2)$. Second, the locations of the term $[x \to \hat{v}_1]\, \hat{t}_2$ are included in the domain of $\hat{m}''$ because the inclusions "$\mathsf{locs}(\hat{v}_1) \subseteq \mathrm{dom}(\hat{m}'')$" and "$\mathsf{locs}(\hat{t}_2) \subseteq \mathrm{dom}(\hat{m})$" and "$\mathrm{dom}(\hat{m}) \subseteq \mathrm{dom}(\hat{m}'')$" hold. (The latter is a consequence of the fact that the store can only grow through time.)

• Case $(\hat{t}_1\, ;\, \hat{t}_2)_{/\hat{m}} \Downarrow^{:} \hat{v}_{/\hat{m}'}$. The treatment of sequences is a simplified version of the treatment of non-polymorphic let-bindings.

• Case $(\mathsf{let}\, x = \hat{w}_1 \,\mathsf{in}\, \hat{t}_2)_{/\hat{m}} \Downarrow^{:} \hat{v}_{/\hat{m}'}$ with $([x \to \hat{w}_1]\, \hat{t}_2)_{/m''} \Downarrow^{:} \hat{v}_{/\hat{m}'}$. Let S_1 be the type of x. Let $\tilde{w}_1$ and $\tilde{t}_2$ be the subterms corresponding to $\hat{w}_1$ and $\hat{t}_2$, respectively. The goal is:

$$\forall X.\quad X = \lceil \alpha\, \tilde{w}_1 \rceil \ \Rightarrow\ [\![\, \alpha\, \tilde{t}_2 \,]\!]^{(x \mapsto X)}\, H\, Q$$

Let X be a value of type $[\![S_1]\!]$ such that X is equal to $\lceil \alpha\, \tilde{w}_1 \rceil$. The goal can be rewritten as "$[\![\, \alpha\, \tilde{t}_2 \,]\!]^{(x \mapsto \lceil \alpha\, \tilde{w}_1 \rceil)}\, H\, Q$". By the substitution lemma, this proposition is equivalent to "$[\![\, \alpha\, ([x \to \tilde{w}_1]\, \tilde{t}_2) \,]\!]^{\emptyset}\, H\, Q$". The conclusion then follows directly from the induction hypothesis.

• Case $(\mathsf{let\,rec}\, f = \Lambda\overline{A}.\lambda x.\hat{t}_1 \,\mathsf{in}\, \hat{t}_2)_{/\hat{m}} \Downarrow^{:} \hat{v}_{/\hat{m}'}$ with $([f \to \mu f.\Lambda\overline{A}.\lambda x.\hat{t}_1]\,\hat{t}_2)_{/\hat{m}} \Downarrow^{:} \hat{v}_{/\hat{m}'}$. Let $\tilde{t}_1$ and $\tilde{t}_2$ be the subterms of $\tilde{t}$ corresponding to the labelling of the subterms $\hat{t}_1$ and $\hat{t}_2$. The proof is very similar to that of the purely-functional setting, so I give only the key steps. The goal is to prove:

$$\mathsf{body}\,(\mu f.\Lambda\overline{A}.\lambda x.(\alpha\,\tilde{t}_1))^{\{F\}} \quad \Rightarrow \quad [\![\,\alpha\,\tilde{t}_2\,]\!]^{(f\mapsto F)}\,H\,Q$$

By the substitution lemma, the characteristic formula $[\![\,\alpha\,\tilde{t}_2\,]\!]^{(f\mapsto F)}$ is equal to $[\![\,[f \to (\mu f.\Lambda\overline{A}.\lambda x.(\alpha\,\tilde{t}_1))^{\{F\}}]\,(\alpha\,\tilde{t}_2)\,]\!]^{\emptyset}$. So, the goal can be reformulated as $[\![\,(\alpha\,([f \to (\mu f.\Lambda\overline{A}.\lambda x.\tilde{t}_1)^{\{F\}}]\,\tilde{t}_2)\,]\!]^{\emptyset}\,H\,Q$. This proposition follows from the induction hypothesis applied to the set $E \cup \{F\}$.

• Case $(\mu f.\Lambda\overline{A}.\lambda x^{T_0}.\hat{t}_1{}^{T_1}\,(\hat{v}_2^{T_2}))^{T}{}_{/\hat{m}} \Downarrow^{:} \hat{v}_{/\hat{m}'}$, where $([f \to \mu f.\Lambda\overline{A}.\lambda x.\hat{t}_1]\,[x \to \hat{v}_2]\,[\overline{A} \to \overline{T}]\,\hat{t}_1)_{/\hat{m}} \Downarrow^{:} \hat{v}_{/\hat{m}'}$, and the assumptions $T_2 = [\overline{A} \to \overline{T}]\,T_0$ and $T = [\overline{A} \to \overline{T}]\,T_1$ hold. Again, the structure of the proof is similar to that of the purely-function setting. Let $\tilde{t}_1$ and $\tilde{v}_2$ be the labelled subterms associated with $\hat{t}_1$ and $\hat{v}_2$, respectively. By hypothesis, the function $\mu f.\Lambda\overline{A}.\lambda x.\alpha\,\tilde{t}_1$ is labelled with some constant F such that the proposition $\mathsf{body}\,(\mu f.\Lambda\overline{A}.\lambda x.\alpha\,\tilde{t}_1)^{\{F\}}$ holds. The goal is to prove "$\mathsf{AppReturns}\,F\,\lceil\alpha\,\tilde{v}_2\rceil\,H\,Q$". This proposition is proved by application of the body hypothesis. The premise to be established is "$[\![\,[\overline{A} \to \overline{T}]\,(\alpha\,\tilde{t}_1)\,]\!]^{(x\mapsto\lceil\alpha\,\tilde{v}_2\rceil, f\mapsto F)}\,H\,Q$". By application of the substitution lemma, and by factorizing the renaming α, the proposition can be reformulated as "$[\![\,\alpha\,([f \to (\mu f.\Lambda\overline{A}.\lambda x.\tilde{t}_1)^{\{F\}}]\,[x \to \tilde{v}_2]\,[\overline{A} \to \overline{T}]\,\tilde{t}_1)\,]\!]^{\emptyset}\,H\,Q$", which is provable directly from the induction hypothesis.

• Case $(\mathsf{if\,true\,then}\,\hat{t}_1\,\mathsf{else}\,\hat{t}_2)_{/\hat{m}} \Downarrow^{:} \hat{v}_{/\hat{m}'}$ with $\hat{t}_{1/\hat{m}} \Downarrow^{:} \hat{v}_{/\hat{m}'}$. The goal is:

$$(\mathsf{true} = \mathsf{true} \Rightarrow [\![\,\alpha\,\tilde{t}_1\,]\!]^{\emptyset}\,H\,Q) \wedge (\mathsf{true} = \mathsf{false} \Rightarrow [\![\,\alpha\,\tilde{t}_2\,]\!]^{\emptyset}\,H\,Q)$$

Thus, it suffices to prove the proposition "$[\![\,\alpha\,\tilde{t}_1\,]\!]^{\emptyset}\,H\,Q$". The proposition follows directly from the induction hypothesis. The case where the argument of the condition is the value false is symmetrical.

• Case $(\mathsf{for}\,x = n\,\mathsf{to}\,n'\,\mathsf{do}\,\hat{t}_1)_{/\hat{m}} \Downarrow^{:} tt_{/\hat{m}'}$. There are two cases. Either $n > n'$ and the loop does not execute. In this case, the goal is to prove $H \rhd Q\,tt$. It suffices to instantiate α' as α and to check that "$\mathsf{mgv}\,\alpha\,tt\,\lceil tt\rceil$" holds. Otherwise, $n \leq n'$. In this case, I prove the characteristic formula for for-loops stated in terms of a loop invariant. This formula is indeed a sufficient condition for establishing the general characteristic formula that supports local reasoning.

The execution of the loop goes through several intermediate states, call them $(\hat{m}_i)_{i\in[n,n'+1]}$, such that $\hat{m}_n = \hat{m}$ and $\hat{m}_{n'+1} = \hat{m}'$. The goal is to find an invariant I that satisfies the three following conjuncts, in which N stands for $\lceil n\rceil$ and N' stands for $\lceil n'\rceil$.

$$\begin{cases} H \rhd I\,N \\ I\,(N'+1) \rhd Q\,\lceil tt\rceil \\ \forall X \in [N, N'].\quad [\![\,\hat{t}_1\,]\!]^{(x\mapsto X)}\,(I\,X)\,(\#\,I\,(X+1)) \end{cases}$$

Let I be the predicate "$\lambda i.\,\exists\alpha'.\,[\alpha' \sqsupseteq \alpha] * (\mathsf{mgh}\,\alpha'\,\hat{m}_i)$". For the first conjunct, the heap predicate H is equal to $(\mathsf{mgh}\,\alpha\,\hat{m}_n)$, so $H \rhd I\,N$ holds. For the second

conjunct, the heap predicate $I\,(N'+1)$ is equivalent to "$\exists\alpha'.\ [\alpha' \sqsupseteq \alpha] * (\mathsf{mgh}\,\alpha'\,\hat{m}')$". It thus entails "$Q\,\lceil tt \rceil$", which is defined as "$\mathsf{mgp}\,tt\,\hat{m}'\,\lceil tt \rceil$".

For the third conjunct, let X be a Coq integer in the range $[N, N']$. Let p be the corresponding program integer, in the sense that $\lceil p \rceil$ is equal to X. By the substitution lemma, the goal "$[\![\,\hat{t}_1\,]\!]^{(x \mapsto X)}\,(I\,X)\,(\#\,I\,(X+1))$" simplifies to "$[\![\,[x \to p]\,\hat{t}_1\,]\!]^{\emptyset}\,(I\,\lceil p \rceil)\,(\#\,I\,(X+1))$". Since the characteristic formula is a local predicate, we can extract from the pre-condition $I\,\lceil p \rceil$ the assumption that there exists a map α'' that extends α and such that "$[\![\,[x \to p]\,\hat{t}_1\,]\!]^{\emptyset}\,(\mathsf{mgh}\,\alpha''\,\hat{m}_p)\,(\#\,I\,(X+1))$" does hold. The induction hypothesis applied to $([x \to p]\,\hat{t}_1)_{/\hat{m}_p} \Downarrow^{:} tt_{/\hat{m}_{p+1}}$ gives "$[\![\,[x \to p]\,\hat{t}_1\,]\!]^{\emptyset}\,(\mathsf{mgh}\,\alpha''\,\hat{m}_p)\,(\mathsf{mgp}\,\alpha''\,tt\,\hat{m}_{p+1})$".

It remains to invoke the rule of consequence and check that the post-condition "$\mathsf{mgp}\,\alpha''\,tt\,\hat{m}_{p+1}$" entails the post-condition $\#\,I\,(\lceil p \rceil + 1)$. The former post-condition asserts the existence of a map α' that extends α'' such that the heap satisfies "$\mathsf{mgh}\,\alpha'\,\hat{m}_{p+1}$". The latter asserts the existence of a map α''' that extends α such that the heap satisfies "$\mathsf{mgh}\,\alpha'''\,\hat{m}_{p+1}$". To prove it, it suffices to instantiate α''' as α' and to check that α' extends α. This fact can be obtained by transitivity, since α' extends α'' and α'' extends α.

• Case $(\mathsf{while}\,\hat{t}_1\,\mathsf{do}\,\hat{t}_2)_{/\hat{m}} \Downarrow^{:} tt_{/\hat{m}'}$. The proof of completeness of characteristic formulae for while-loops generalizes the proof given for for-loops. Again, I rely on a loop invariant. The evaluation of the loop "$(\mathsf{while}\,\hat{t}_1\,\mathsf{do}\,\hat{t}_2)$" goes through a sequence of intermediate typed states:

$$\hat{m} = \hat{m}_0, \quad \hat{m}'_0, \quad \hat{m}_1, \quad \hat{m}'_1, \quad \hat{m}_2, \quad \hat{m}'_2, \quad \ldots, \quad \hat{m}_n, \quad \hat{m}'_n = \hat{m}'$$

where the evaluation of the loop condition $\hat{t}_1$ takes from a state $\hat{m}_i$ to the state $\hat{m}'_i$ and the evaluation of the loop body $\hat{t}_2$ takes from a state $\hat{m}'_i$ to the state $\hat{m}_{i+1}$. So, $\hat{m}_i$ describes a state before the evaluation of the loop condition $\hat{t}_1$ and $\hat{m}'_i$ describes the corresponding state right after the evaluation of $\hat{t}_1$. Thanks to the bijection between well-typed memory stores and heaps, I can define a sequence of heaps h_i and h'_i that correspond to $\hat{m}_i$ and $\hat{m}'_i$.

In the characteristic formula for while-loops expressed with loop invariants, I instantiate A as the type Heap, and I instantiate $\prec$ as a binary relation such that $h_i \prec h_j$ holds if and only if i is greater than j. Moreover, I instantiate the predicates I and J as follows.

$$\begin{array}{rcl} I & \equiv & \lambda h.\ \ \exists i.\, [h = h_i] * \exists\alpha'.\, [\alpha' \sqsupseteq \alpha] * (\mathsf{mgh}\,\alpha'\,\hat{m}_i) \\ J & \equiv & \lambda h b.\, \exists i.\, [h = h_i] * \exists\alpha'.\, [\alpha' \sqsupseteq \alpha] * (\mathsf{mgh}\,\alpha'\,\hat{m}'_i) * [b = \mathsf{true} \Leftrightarrow i < n] \end{array}$$

It remains to establish the following facts.

$$\left\{\begin{array}{l} \mathsf{well\text{-}founded}(\prec) \\ \exists X_0.\ H \rhd I\,X_0 \\ \forall X.\ [\![\hat{t}_1]\!]\,(I\,X)\,(J\,X) \\ \forall X.\ [\![\hat{t}_2]\!]\,(J\,X\,\mathsf{true})\,(\#\,\exists Y.\ (I\,Y) * [Y \prec X]) \\ \forall X.\ J\,X\,\mathsf{false} \rhd Q\,\lceil tt \rceil \end{array}\right.$$

(1) The relation $\prec$ is well-founded because no two memory states $\hat{m}_i$ and $\hat{m}_j$ can be equal when $i \neq j$. Otherwise, if two states were equal, then the while-loop would run as an infinite loop, contradicting the assumption that the execution of the term $\hat{t}$ does terminate.

(2) The value X_0, of type Heap, can be instantiated as h_0. By hypothesis, this heap satisfies "$\mathsf{mgh}\,\alpha\,\hat{m}_0$", so it satisfies the heap predicate $I\,h_0$ (take $i = 0$ and $\alpha' = \alpha$ in the definition of I).

(3) Let X be a heap. Consider the goal $[\![t_1]\!]\,(I\,X)\,(J\,X)$. Let h be another name for X. Unfolding the definition of I and exploiting the fact that $[\![t_1]\!]$ is a local predicate, it suffices to prove:

$$\forall i.\, \forall \alpha'.\, h = h_i \wedge \alpha' \sqsupseteq \alpha \;\Rightarrow\; [\![\hat{t}_1]\!]\,(\mathsf{mgh}\,\alpha'\,\hat{m}_i)\,(J\,h)$$

By definition of the heaps $\hat{m}_i$ and $\hat{m}'_i$, the reduction $\hat{t}_{1/\hat{m}_i} \Downarrow^{\cdot} \lfloor r \rfloor_{/\hat{m}'_i}$ holds, where r is a boolean such that "$r = \mathsf{true} \Leftrightarrow i < n$". By induction hypothesis, $\hat{t}_1$ admits the post-condition "$\mathsf{mgp}\,b\,\hat{m}'_i$", which is equivalent to "$\lambda b.\, \exists \alpha''.\, [\alpha'' \sqsupseteq \alpha'] * [b = r] * (\mathsf{mgh}\,\alpha''\,\hat{m}'_i)$". One can check that this post-condition entails $J\,h$ (instantiate i as i, α' and α'' and in the definition of J).

(4) Let X be a heap. Consider the goal $[\![t_2]\!]\,(J\,X\,\mathsf{true})\,(\#\,\exists Y.\ (I\,Y) * [Y \prec X])$. By definition of J, X is equal to a heap h_i with $i < n$. Moreover, there exists a map α' that extends α such that the input heap satisfies $\mathsf{mgh}\,\alpha'\,\hat{m}'_i$. The induction hypothesis applied to the reduction sequence $\hat{t}_{2/\hat{m}'_i} \Downarrow^{\cdot} tt_{/\hat{m}_{i+1}}$ gives the proposition "$[\![\,\hat{t}_2\,]\!]^{\emptyset}\,(\mathsf{mgh}\,\alpha'\,\hat{m}'_i)\,(\mathsf{mgp}\,\alpha'\,tt\,\hat{m}_{i+1})$". The post-condition "$\mathsf{mgp}\,\alpha'\,tt\,\hat{m}_{i+1}$" asserts the existence of a map α'' that extends α' such that the output heap satisfies $\mathsf{mgh}\,\alpha''\,\hat{m}_{i+1}$. So, this post-condition entails target post-condition $\#\,\exists Y.\ (I\,Y) * [Y \prec X]$. Indeed, it suffices to instantiate Y as h_{i+1}, which is indeed smaller than h_i with respect to $\prec$. The heap predicate $I\,Y$ is then equivalent to "$\exists \alpha'.\, [\alpha' \sqsupseteq \alpha] * (\mathsf{mgh}\,\alpha'\,\hat{m}_{i+1})$", which follows from $\mathsf{mgh}\,\alpha''\,\hat{m}_{i+1}$ by instantiating α' as α''. The relation $\alpha'' \sqsupseteq \alpha$ is obtained by transitivity.

(5) It remains to prove "$J\,X\,\mathsf{false} \rhd Q\,tt$" for any heap X. By definition of J, there exists an index i such that X is equal to h_i and i is not smaller than n. Hence, X must be equal to h_n. So, $J\,X\,\mathsf{false}$ is equivalent to $\exists \alpha'.\, [\alpha' \sqsupseteq \alpha] * \mathsf{mgh}\,\alpha'\,\hat{m}'_i$. This heap predicate is indeed equivalent to $Q\,tt$, by definition of Q.

• Case $(\mathsf{ref}\,\hat{v})^{|\mathsf{oc}}{}_{/\hat{m}} \Downarrow^{\cdot} l^{|\mathsf{oc}}{}_{/\hat{m}'}$ where l is a location fresh from the domain of $\hat{m}$ and $\hat{m}'$ is the heap $\hat{m} \uplus [l \mapsto \hat{v}]$. Let T denote the type of $\hat{v}$, and let $\mathcal{T}$ denote the Coq type $\lceil T \rceil$. Let $\tilde{v}$ be the labelled value associated with $\hat{v}$, and let V be a shorthand for $\lceil \alpha\,\tilde{v} \rceil$ The goal is to prove "$\mathsf{AppReturns}_{\mathcal{T},|\mathsf{oc}}\,\mathsf{ref}\,\lceil \alpha\,\tilde{v} \rceil\,H\,Q$".

The specification of ref gives "$\mathsf{AppReturns}_{\mathcal{T},|\mathsf{oc}}\,\mathsf{ref}\,V\,[\,]\,(\lambda L.\, L \hookrightarrow_{\mathcal{T}} V)$". The frame rule then gives "$\mathsf{AppReturns}_{\mathcal{T},|\mathsf{oc}}\,\mathsf{ref}\,V\,H\,(\lambda L.\, (L \hookrightarrow_{\mathcal{T}} V) * H)$". The goal follows from the rule of consequence applied to this fact. It remains to establish that, for any location L, the heap predicate "$(L \hookrightarrow_{\mathcal{T}} \hat{v}) * H$" is stronger than $Q\,L$. The heap predicate $Q\,L$ is equivalent to:

$$\exists \alpha'.\, [\alpha' \sqsupseteq \alpha] * [\mathsf{mgv}\,\alpha'\,l\,L] * (\mathsf{mgh}\,\alpha'\,\hat{m}')$$

Let L be a location and h' be a heap satisfying the predicate $(L \hookrightarrow_{\mathcal{T}} \hat{v}) * H$. The goal is to prove that the above heap predicate also applies to h'. Then, to prove the goal, I instantiate α' as $\alpha \uplus [l \mapsto \lfloor L \rfloor]$. The first subgoal $\alpha' \sqsupseteq \alpha$ holds by definition of α'. The second subgoal "$\mathsf{mgv}\,\alpha'\,l\,L$" simplifies to "$L = \alpha'\,l$", which is also true by definition of α'. It remains to prove the third subgoal, which asserts that h' satisfies the heap predicate $(\mathsf{mgh}\,\alpha'\,\hat{m}')$.

Because h' satisfies $(L \hookrightarrow_{\mathcal{T}} \hat{v}) * H$, the heap h' decomposes as a singleton heap satisfying $(L \hookrightarrow_{\mathcal{T}} \hat{v})$ and as a heap h satisfying H, which was defined as "$\mathsf{mgh}\,\alpha\,\hat{m}$". The assumptions given by "$\mathsf{mgh}\,\alpha\,\hat{m}\,h$" are as follows.

$$\begin{cases} \mathrm{dom}(h) = \mathrm{dom}(\alpha\,\hat{m}) \\ \forall l' \in \mathrm{dom}(\hat{m}).\ \mathsf{mgv}\,\alpha\,(\hat{m}[l'])\,V' \qquad \text{where } (\mathcal{T}', V') = h[\alpha\,l'] \\ \mathsf{locs}(\hat{m}) \subseteq \mathrm{dom}(\alpha) \end{cases}$$

To prove $(\mathsf{mgh}\,\alpha'\,\hat{m}'\,h')$, we need to establish the following facts.

$$\begin{cases} \mathrm{dom}(h') = \mathrm{dom}(\alpha'\,\hat{m}') \\ \forall l' \in \mathrm{dom}(\hat{m}').\ \mathsf{mgv}\,\alpha'\,(\hat{m}'[l'])\,V' \qquad \text{where } (\mathcal{T}', V') = h'[\alpha'\,l'] \\ \mathsf{locs}(\hat{m}') \subseteq \mathrm{dom}(\alpha') \end{cases}$$

The first and the third facts are easy to verify. For the second fact, let l' be a location in the domain of $\hat{m}'$. If l' is equal to l, then $(\alpha\,l')$ is equal to L and the goal is "$\mathsf{mgv}\,\alpha'\,(\hat{m}'[l])\,V$". The proposition simplifies to "$\mathsf{mgv}\,\alpha'\,\hat{v}\,(\alpha\,\tilde{v})$", which follows from the tautology "$\mathsf{mgv}\,\alpha\,\hat{v}\,(\alpha\,\tilde{v})$", by covariance of mgs in the α-renaming map. Otherwise, if l' is not equal to l, then l' belongs to the domain of $\hat{m}$ and the covariance of mgv in the α-renaming map can be used to derive "$\mathsf{mgv}\,\alpha'\,(\hat{m}[l'])\,V'$" from "$\mathsf{mgv}\,\alpha\,(\hat{m}[l'])\,V'$".

• Case $(\mathsf{get}\,l)^{T}{}_{/\hat{m}} \Downarrow^{:} \hat{v}^{T}{}_{/\hat{m}}$ where $\hat{v}^T = \hat{m}[l]$. Let $\mathcal{T}$ stand for $[\![T]\!]$ and let L stand for $\lceil \alpha\,l \rceil$. The goal is "$\mathsf{AppReturns}_{\mathsf{loc},\mathcal{T}}\,\mathsf{get}\,L\,(\mathsf{mgh}\,\alpha\,\hat{m})\,(\mathsf{mgp}\,\alpha\,\hat{v}\,\hat{m})$". We could use the frame rule like in the previous proof case to prove this goal, however the proof is simpler when we directly unfold the definition of $\mathsf{AppReturns}$ and work with the predicate $\mathsf{AppEval}$. The goal then reformulates as follows.

$$\forall h_i\,h_k.\ \mathsf{mgh}\,\alpha\,\hat{m}\,h_i \ \Rightarrow\ \exists V\,h_f\,h_g.\ \begin{cases} \mathsf{AppEval}\,\mathsf{get}\,L\,(h_i + h_k)\,V\,(h_f + h_k + h_g) \\ \mathsf{mgp}\,\alpha\,\hat{v}\,\hat{m}\,V\,h_f \end{cases}$$

From the specification of get applied to the location L, we get:

$$\forall V.\ \ \mathsf{AppReturns}_{\mathsf{loc},\mathcal{T}}\,\mathsf{get}\,L\,(L \hookrightarrow_{\mathcal{T}} V)\,(\lambda V'.\,[V' = V] * (L \hookrightarrow_{\mathcal{T}} V))$$

Reformulating this proposition in terms of $\mathsf{AppEval}$ gives:

$$\forall V\,h_i'\,h_k'.\ (L \hookrightarrow_{\mathcal{T}} V)\,h_i' \ \Rightarrow\ \exists V'\,h_f'\,h_g'.\ \begin{cases} \mathsf{AppEval}\,\mathsf{get}\,L\,(h_i' + h_k')\,V'\,(h_f' + h_k' + h_g') \\ V' = V \\ (L \hookrightarrow_{\mathcal{T}} V')\,h_f' \end{cases}$$

It remains to find the suitable instantiations for proving the goal by exploiting the above proposition. Since l belongs to the domain of $\hat{m}$ and since the hypothesis $\mathsf{mgh}\,\alpha\,\hat{m}\,h_i$ holds, we know that L belongs to the domain of h_i and that $\mathsf{mgv}\,\alpha\,(\hat{m}[l])\,(h_i[l])$ holds. Let V stand for the value $h_i[l]$. The assertion reformulates as $\mathsf{mgv}\,\alpha\,\hat{v}\,V$. Moreover, there exists a heap h_r such that the heap h_i decomposes as the disjoint union of the singleton heap $L \rightarrow_{\mathcal{T}} V$ and of h_r. Let h'_i be instantiated as $L \rightarrow_{\mathcal{T}} V$, and let h'_k be instantiated as $h_r + h_k$. The union $h'_i + h'_k$ equal to the union $h_i + h_k$.

We can check the premise $(L \hookrightarrow_{\mathcal{T}} V)\,h'_i$, and obtain the existence of V', h'_f and h'_g satisfying the three conclusions derived from the specification of get. The second conclusion ensures that V' is equal to V. The third conclusion, $(L \hookrightarrow_{\mathcal{T}} V)\,h'_f$ asserts that h'_f is a singleton heap of the form $L \rightarrow_{\mathcal{T}} V$, so it is the same as h'_i. In particular, the union $h'_f + h'_k$ is equal to the union $h_i + h_k$. The first conclusion, $\mathsf{AppEval}\,\mathsf{get}\,L\,(h'_i + h'_k)\,V'\,(h'_f + h'_k + h'_g)$, is then equivalent to the proposition $\mathsf{AppEval}\,\mathsf{get}\,L\,(h_i + h_k)\,V\,(h_i + h_k + h'_g)$.

To conclude, it remains to instantiate V as V, h_f as h_i, h_g as h'_g, and to prove $\mathsf{mgp}\,\alpha\,\hat{v}\,\hat{m}\,V\,h_i$. Instantiating α' as α in this proposition, we have to prove $\mathsf{mgh}\ \alpha\hat{m}\,h_i$, which holds by assumption, and to prove $\mathsf{mgv}\,\alpha\,\hat{v}\,V$, a fact which has been extracted earlier on from the assumption $\mathsf{mgh}\,\alpha\,\hat{m}\,h_i$.

• Case $(\mathsf{set}\,l\,\hat{v})^{\mathsf{unit}}{}_{/\hat{m}} \Downarrow^{:} tt^{\mathsf{unit}}{}_{/\hat{m}'}$ where l belongs to the domain of l and $\hat{m}'$ stands for $\hat{m}[l \mapsto \hat{v}]$. Let T be the type of $\hat{v}$ and let $\mathcal{T}$ stand for $[\![T]\!]$. By assumption, there exists a labelled value $\tilde{v}$ that corresponds to $\hat{v}$, such that $\mathsf{labels}\ E\ \hat{v}$. Let V stand for $\lceil \alpha\,\tilde{v} \rceil$, and let L stand for $\lceil \alpha\,l \rceil$. The goal is:

$$\mathsf{AppReturns}_{(\mathsf{loc}\times\mathcal{T}),\mathsf{unit}}\,\mathsf{set}\,(L,V)\,(\mathsf{mgh}\,\alpha\,\hat{m})\,(\mathsf{mgp}\,\alpha\,tt\,\hat{m})$$

As in the previous proof case, we can rewrite this goal in terms of AppEval. The statement shown next takes into account the fact that the return value is the unit value.

$$\forall h_i\,h_k.\ \mathsf{mgh}\,\alpha\,\hat{m}\,h_i\ \Rightarrow\ \exists h_f\,h_g.\ \left\{\begin{array}{l}\mathsf{AppEval}\,\mathsf{set}\,(L,V)\,(h_i+h_k)\,tt\,(h_f+h_k+h_g)\\ \mathsf{mgp}\,\alpha\,tt\,\hat{m}'\,\lceil tt \rceil\,h_f\end{array}\right.$$

From the specification of set applied to the location L, we get:

$$\forall V\,\mathcal{T}'\,V'.\ \ \mathsf{AppReturns}_{(\mathsf{loc}\times\mathcal{T}),\mathsf{unit}}\,\mathsf{set}\,(L,V)\,(L \hookrightarrow_{\mathcal{T}'} V')\,(\#\,(L \hookrightarrow_{\mathcal{T}'} V))$$

Reformulating this proposition in terms of AppEval gives:

$$\begin{array}{l}\forall V\,\mathcal{T}'\,V'\,h'_i\,h'_k.\\ \quad (L \hookrightarrow_{\mathcal{T}'} V')\,h'_i\end{array}\ \Rightarrow\ \exists h'_f\,h'_g.\ \left\{\begin{array}{l}\mathsf{AppEval}\,\mathsf{set}\,(L,V)\,(h'_i+h'_k)\,tt\,(h'_f+h'_k+h'_g)\\ (L \hookrightarrow_{\mathcal{T}} V)\,h'_f\end{array}\right.$$

It remains to find the suitable instantiations. Since l belongs to the domain of $\hat{m}$ and since the hypothesis $\mathsf{mgh}\,\alpha\,\hat{m}\,h_i$ holds, we know that L belongs to the domain of h_i and that $\mathsf{mgv}\,\alpha\,(\hat{m}[l])\,(h_i[l])$. Let V' denote the value $h_i[l]$. There exists a heap h_r such that the heap h_i decomposes as the disjoint union of the singleton heap

$L \rightarrow_{\mathcal{T}'} V'$ and of h_r. Let h'_i be instantiated as $L \rightarrow_{\mathcal{T}'} V'$, and let h'_k be instantiated as $h_r + h_k$. The union $h'_i + h'_k$ equal to the union $h_i + h_k$.

We can check the premise $(L \hookrightarrow_{\mathcal{T}'} V')\,(L \rightarrow_{\mathcal{T}'} V')$, and obtain the existence of h'_f and h'_g satisfying the two conclusions derived from the specification of set. The second conclusion, $(L \hookrightarrow_{\mathcal{T}} V)\,h'_f$ asserts that h'_f is a singleton heap of the form $L \rightarrow_{\mathcal{T}} V$. Let h_f be instantiated as $h'_f + h_r$. In particular, the union $h'_f + h'_k$ is equal to the union $h_f + h_k$. Moreover, let h_g be instantiated as h'_g. The first conclusion, $\mathsf{AppEval}\,\mathsf{get}\,L\,(h'_i + h'_k)\,V'\,(h'_f + h'_k + h'_g)$ is then equivalent to $\mathsf{AppEval}\,\mathsf{get}\,L\,(h_i + h_k)\,V\,(h_f + h_k + h_g)$.

It remains to prove $\mathsf{mgp}\,\alpha\,tt\,\hat{m}'\,\lceil tt \rceil\,h_f$. Instantiating α' as α in this predicate, it remains to prove $\mathsf{mgv}\,\alpha\,tt\,tt$, which is true, and $\mathsf{mgh}\,\alpha\,\hat{m}'\,h_f$. To prove the latter, let l' be a location in the domain of $\hat{m}'$, which is the same as the domain of $\hat{m}$. We have to prove $\mathsf{mgv}\,\alpha\,(\hat{m}[l'])\,(h_f[\lceil \alpha\,l' \rceil])$. There are two cases. If l' is not the location l, then $\hat{m}'[l']$ is equal to $\hat{m}[l']$ and $h_f[\lceil \alpha\,l' \rceil]$ is equal to $h_i[\lceil \alpha\,l' \rceil]$, so the assumption "$\mathsf{mgh}\,\alpha\,\hat{m}\,h_i$" can be used to conclude. Otherwise, if l' is the location l, then $\hat{m}'[l]$ is equal to $\hat{v}$ and $h[\lceil \alpha\,l \rceil]$ is equal to V (because V stands for $h[L]$ and L has been defined as $\lceil \alpha\,l \rceil$), so the goal is to prove $\mathsf{mgv}\,\alpha\,\hat{v}\,V$, which holds because V has been defined $\lceil \alpha\,\tilde{v} \rceil$ and $\mathsf{labels}\,E\,\hat{v}$ holds by assumption.

• Case $(\mathsf{cmp}\,l\,k')_{/\hat{m}} \Downarrow^{:} \hat{b}^{\mathsf{bool}}{}_{/\hat{m}}$. The proof is straightforward since cmp does not involve any reasoning on the heap. □

7.3.3 Completeness for integer results

A simpler statement of the completeness theorem can be given in the case of a program that admits the type int in ML. To lighten the presentation, I identify Caml integers with Coq integers.

Theorem 7.3.2 (Completeness for ML programs with integer results) *Consider a closed ML program of type* int. *Let* $\hat{t}$ *denote the corresponding term typed in weak-ML, and let* t *denote the corresponding raw term. Let* n *be an integer, let* P *be a Coq predicate on integers, and let* m *be a memory store.*

$$t_{/\emptyset} \Downarrow n_{/m} \quad \wedge \quad P\,n \quad \Rightarrow \quad [\![\,\hat{t}\,]\!]^{\emptyset}\,[\,]\,(\lambda n.\,[P\,n])$$

Proof Since the term t is well-typed in ML, the reduction derivation $t_{/\emptyset} \Downarrow n_{/m}$ can be turned into a typed reduction derivation $\hat{t}^{\mathsf{int}}{}_{/\emptyset} \Downarrow^{:} n^{\mathsf{int}}{}_{/\hat{m}}$. By the completeness theorem applied to that derivation, there exists a renaming α such that:

$$[\![\,\hat{t}\,]\!]^{\emptyset}\,[\,]\,(\lambda V.\,\exists\alpha.\,[\mathsf{mgv}\,\alpha\,n\,V] * (\mathsf{mgh}\,\alpha\,m))$$

The characteristic formula is a local predicate, so we can apply the rule of garbage collection to ignore the post-condition about the final store m. It gives:

$$[\![\,\hat{t}\,]\!]^{\emptyset}\,[\,]\,(\lambda V.\,\exists\alpha.\,[\mathsf{mgv}\,\alpha\,n\,V])$$

To prove the conclusion, we apply the rule of consequence, and there remains to establish an implication between the post-conditions:

$$\forall V.\ (\exists \alpha.\, \mathsf{mgv}\, \alpha\, n\, V) \ \Rightarrow\ P\, V$$

The hypothesis "$\mathsf{mgv}\, \alpha\, n\, V$" asserts the existence of a correct labelling $\tilde{n}$ of n such that $V = \lceil \alpha\, \tilde{n} \rceil$. Since the integer n does not contain any function nor any location, the hypothesis simplifies to $V = n$. The conclusion $P\, V$ then follows directly from the assumption $P\, n$. $\square$

Chapter 8

Related work

The discussion of related work is organized around five sections. The first one is concerned with the origins of characteristic formulae and the comparison with the program logics developed by Honda, Berger, and Yoshida. In a second part, I focus on Separation Logic. I then discuss approaches based on Verification Condition Generators (VCG), approaches based on shallow embeddings, and approaches based on deep embeddings.

8.1 Characteristic formulae

Characteristic formulae in process calculi The notion of characteristic formula originates in process calculi. The characterization theorem [70, 57] states that two processes are bisimilar if and only if they satisfy the same set of formulae in Hennessy-Milner logic [35]. In this context, a formula F is said to be the *characteristic formula* of a process p if the set of processes that satisfy F matches exactly the set of processes that are bisimilar to p. Graf and Sifakis [32] described an algorithm for constructing the characteristic formulae of a process from the syntactic definition of that process. More precisely, they explained how to build a modal logic formula that characterizes the equivalence class of a given CCS process (for processes that do not involve recursion). Aceto and Ingólfsdóttir [1] later described a similar generation algorithm for translating regular CCS processes into their characteristic formulae, which are there expressed in Hennessy-Milner logic extended with recursion.

The behavioral equivalence or dis-equivalence of two processes can be established by comparing their characteristic formulae. Such proofs can be conducted in a high-level temporal logic rather than through reasoning on the syntactic definition of the processes. Somewhat similarly, the characteristic formulae developed in this thesis allow reasoning on a program to be conducted on higher-order logic formulae, without referring to the source code of that program at any time.

Total characteristic assertion pairs The first development of a program-logic counterpart to process-logic characteristic formulae is due to recent work by Honda, Berger and Yoshida [40]. This work originates in the study of a correspondence between process logics and program logics [38], where Honda showed how an encoding of a high-level programming language into the π-calculus can be used to turn a logic for the π-calculus with linear types into a compositional program logic for a higher-order functional programming language. This program logic, which is described in detail in [41], was later extended to an imperative programming language with global state [10], and then further extended to support reasoning on aliasing [39] and reasoning on local state [87].

In the logics for higher-order imperative programs, the specification judgment take the form of a Hoare triple $\{C\}\ t :_x \{C'\}$, where t is a term, C and C' are assertions about the initial heap and the final heap, and x is a bound name used to refer to the result produced by t in the post-condition C'. So, the interpretation of the judgment $\{C\}\ t :_x \{C'\}$ is essentially the same as that of the triple $\{C\}\ t\ \{\lambda x.\, C'\}$, which follows the presentation that I have used so far. The specification of functions and higher-order functions involves a ternary predicate, called evaluation formula and written "$v_1 \bullet v_2 = v_3$", which asserts that the application of the value v_1 to the value v_2 returns the value v_3. The specification thus takes place in a first-order logic extended with this ad-hoc evaluation predicate. Another specificity of the assertion logic is that its values are the values of the programming language (i.e., PCF values), including in particular non-terminating functions. Moreover, in the assertion logic, equality is interpreted as observational equivalence. All those constructions make the assertion logic nonstandard, making it difficult to reuse existing proof tools.

The series of work on program logics by Honda, Berger and Yoshida culminated in the development of *total characteristic assertion pairs* [40], abbreviated as TCAP, which corresponds to the notion of most-general Hoare triple. A most-general Hoare triple, also called most-general formula, is made of the weakest pre-condition, which corresponds to the minimal requirement for safe execution, and of the strongest post-condition, which precisely relates the output heap to the input heap. Most-general formulae were introduced by Gorelick in 1975 [31] for establishing a completeness result for Hoare logic. Yet, this theoretical result does not help verify programs in practice, because most-general formulae were there expressed in terms of the syntax and the semantics of the source program. The total characteristic assertion pairs (TCAPs) developed by Honda, Berger and Yoshida [40] precisely solve that problem, as TCAPs express the weakest pre-condition and the strongest post-condition in the assertion logic, without any reference to program syntax. Moreover, the TCAP of a program can be constructed automatically by application of a simple algorithm.

As those researchers point out, TCAPs suggest a new approach to proving that a program satisfies a given specification. Indeed, rather than building a derivation using the reasoning rules of the Hoare program logic, one may simply prove that the pre-condition of the specification implies the weakest pre-condition and that the post-condition of the specification is implied by the strongest post-condition. The verification of those two implications can be conducted entirely at the logical

level. Yet, this appealing idea was never implemented, mainly because the treatment of first-class functions and of evaluation formulae in the assertion logic makes the logic nonstandard and precludes the use of a standard theorem prover. A central contribution of this thesis has been to show how to reconcile first-class functions and evaluation formulae with a standard logic, by representing functions in the logic through the deep embedding of their code (with the type Func) and by defining an evaluation formula (the predicate AppEval) in terms of the deep embedding of the semantics of the source programming language.

The presentation of characteristic formulae that I build also differ in several ways from TCAPs. First, characteristic formulae rely on existential quantification of intermediate specifications and loop invariants. This was not the case in TCAPs, which are expressed in first-order logic. Second, I have shown that characteristic formulae could be pretty-printed just like source code. This possibility was not pointed out in the work on TCAPs, even though, in principle, it should be possible to apply a similar notation system to TCAPs. Third, TCAPs rely on a reachability predicate for reasoning on dynamic allocation, whereas I have based on my work upon standard technique from Separation Logic, whose effectiveness has already been demonstrated. In particular, I introduced the predicate local to integrate the frame rule directly inside characteristic formulae.

8.2 Separation Logic

Tight specifications and the frame rule Separation Logic has been developed initially by Reynolds [77] and by O'Hearn and Yang [67, 86], and subsequently by many others. It builds on earlier work by Burstall [14], on work by Bornat [12], and on the logic of Bunched Implications developed by O'Hearn, Ishtiaq and Pym [66, 42]. The starting point of Separation Logic is the tight interpretation of Hoare triples. The tight interpretation of a triple $\{H\}\ t\ \{Q\}$ asserts that the execution of the term t is only allowed to modify memory cells that are described by the pre-condition H. This interpretation is the key to the soundness of the frame rule, which enables local reasoning.

Following the traditional presentation of Hoare logic, Separation Logic allows the local variables of a program to be modified, even though it does not require local variables to be described with an assertion of the form $x \hookrightarrow v$. In other words, in Separation Logic assertions, the name of a local variable is confused with the current value of that variable. For this reason, the frame rule is usually presented with a side-condition, as shown below. In this rule, c denotes a command, and p, q and r are assertions about the heap.

$$\frac{\{p\}\ c\ \{q\}}{\{p * r\}\ c\ \{q * r\}} \quad \left(\begin{array}{l} \textit{where no variable occurring} \\ \textit{free in } r \textit{ is modified by } c. \end{array} \right)$$

Characteristic formulae require a frame rule without side condition because they cannot refer to program syntax. As observed by O'Hearn [68], the side-condition is

not needed in a language such as Caml, where mutation is only allowed for heap-allocated data. More generally, it would suffice to distinguish the name of local variables from their contents in order to avoid the side-condition in the frame rule.

Separation Logic tools A number of verification tools have been built upon ideas from Separation Logic. For example, Smallfoot, developed by Berdine, Calcagno and O'Hearn [8], is an automated verification tool specialized for programs manipulating linked data structures. Also, Yang *et al* [85] have developed an automated tool for verifying dozens of thousands of lines of source code for device drivers, which mainly involve linked lists.

Because fully-automated tools are inherently limited in scope, other researchers have investigated the possibility to exploit Separation Logic inside interactive proof assistants. The first such embedding of Separation Logic appears to be the one by Marti and Affeldt [52, 53]. Yet, these researchers conducted their Coq proofs by unfolding the definition of the separating conjunction, which breaks the nice level of abstraction brought by this operator. Appel [2] has shown how to build reasoning tactics that do not break the abstraction layer, obtaining significantly-shorter proof scripts. Further Separation Logic tactics were subsequently developed in many research projects, e.g., [24, 82, 62, 17, 55, 59].

Most of those tactics exploit an hypothesis of the form $H\,h$ for proving a goal of the form $H'\,h$. One exception is a recent version of Ynot [17], where the tactics directly handles goals of the form $H \rhd H'$, without involving a heap variable h in the proof. This approach allows working entirely at the level of heap predicates and it allows for the development of more efficient tactics. This is why I have followed that approach in the implementation of my tactic `hsimpl` (§5.7.1).

Local reasoning and while loops The traditional Hoare logic reasoning rule for while loops is expressed using loop invariants. This rule can be used in the context of Separation Logic. However, it does not take full advantage of local reasoning (as explained in §3.3). I came accross the limitation of loop-invariants, found a way to address it, and then learnt that Tuerk had made the same observation and discussed it in a workshop paper [83]. I present his solution and then compare it with mine.

In Tuerk's presentation, the deduction rule that *does* involve a loop invariant is as shown below. In this partial-correctness reasoning rule, b is a boolean condition, c is a set of commands, and I is an invariant.

$$\frac{\{b \wedge I\}\ c\ \{I\}}{\{I\}\ (\mathsf{while}\, b\, \mathsf{do}\, c)\ \{\neg b \wedge I\}}$$

His generalized rule is shown next. There, P denotes a predicate (the invariant becomes parameterized by an index), and x and y denote indices for P. Observe that the premise involves a quantification over all the programs c' that admit a given specification. The variable c' intuitively denotes the remaining iterations of the loop.

So, the sequence $(c\,;\,c')$ corresponds to the execution of the first iteration followed with that of the remaining iterations.

$$\frac{\forall x.\,\forall c'.\,\big(\forall y.\,\{P\,y\}\;c'\;\{\neg b \wedge P\,y\}\big)\;\;\Rightarrow\;\;\{b \wedge P\,x\}\;(c\,;\,c')\;\{\neg b \wedge P\,x\}}{\forall x.\,\{P\,x\}\;(\mathsf{while}\,b\,\mathsf{do}\,c)\;\{\neg b \wedge P\,x\}}$$

The rule involves a negative occurrence of a Hoare triple, so it cannot be used as an inductive definition. Instead, the reasoning rule is presented as a lemma proved correct with respect to the operational semantics of the source language.

The quantification over c' plays a very similar role as the quantification over a variable R involved in the characteristic formulae for while loops. My formulation is slightly more concise because I quantify directly over a given behavior R instead of quantifying over all the programs c' that admit a given behavior. It is not surprising that my solution shares strong similarities with Tuerk's solution, since we both used an encoding of while loops as recursive functions as a starting point for deriving a local-reasoning–friendly version of the reasoning rule for loops.

8.3 Verification Condition Generators

A 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. The generation of proof obligations typically relies on the computation of weakest pre-conditions Following this approach pioneered by Floyd [29], Hoare [36] and Dijkstra [22], a large number of VCGs targeting various programming languages have been implemented in the last decades, including VCGs for full-blown industrial programming languages. For example, Spec-# [5] supports verification of properties about C# programs. Programs are translated into the Boogie [4] intermediate language, from which proof obligations are generated. The SMT solver Z3 [21] is then used to discharge those proof obligations. Most SMT solvers only cope with first-order logic, thus the specification language does not benefit from the expressiveness, modularity, and elegance of higher-order logic. Several researchers have investigated the possibility of extending the VCG approach to a specification language based on higher-order logic. Three notable lines of work are described next.

The Why platform The tool Why, developed by Filliâtre [26], is an intermediate purely-functional language annotated with higher-order logic specification and invariants, on which weakest pre-conditions can be computed for generating proof obligations. The tool is intended to be used in conjunction with a front-end, like Caduceus [27] for C programs or Krakatoa [51] for Java programs. Proof obligations that are not verified automatically by at an SMT solver can be discharged using an interactive proof assistant such as Coq.

However, in practice, those proof obligations are often large and clumsy. This is in part due to the memory model, which does not take advantage of Separation Logic. Moreover, interactive proofs of verification conditions are generally quite

brittle because the proof obligations are quite sensitive to changes in either the source code or its invariants. So, although interactive verification of programs with higher-order logic specifications is made possible by the tool Why, it involves a significant cost. As a consequence, the verification of large programs with Why appears to be well-suited only for programs that require a small number of interactive proofs.

With characteristic formulae, a slightly smaller degree of automation is achievable than with a VCG, however proof obligations always remain tidy and can be easily related to the point of the program they arise from. Characteristic formulae offer some possibility for partially automating the reasoning, through calls to the tactic `xgo` and through the invokation of decision procedures from Coq. When reasoning on the code becomes more involved and requires a lot of intervention from the user, the verification of the code can be conducted step-by-step in a fully-interactive manner. Moreover, by investing a little extra effort in explicitly naming the hypotheses that need to be referred to in the proof scripts, one can build very robust proof scripts.

Jahob The tool Jahob [88], mainly developed by Zee, Kuncak and Rinard, targets the verification of programs written in a subset of Java, and accommodates specifications expressed in higher-order logic. The tool has been successfully employed to verify mutable linked data structures, such as lists and trees. 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 including proof hints directly inside the source code. Those hints are used to guide automated theorem provers, in particular by indicating how existential variables should be instantiated. Although it is not clear how this approach would extend beyond the verification of list and set data structures for which powerful decision procedures can be devised, the programs verified in Jahob exhibit both high-level specifications and an impressive degree of automation.

Yet, the impressively-small number of hints written in a program often covers up the extensive amount of time required for coming up with the appropriate hints. The development process involves running the tool, waiting for the output of automated theorem provers, reading the proof obligations to try and understand why proof obligations could not be discharged automatically, and finally starting over with a slightly different proof hint. This process turns out to be quite time-consuming. Moreover, guessing what hints may work seems to require a good understanding of the working of the automated theorem provers and of the encoding of higher-order logic that is applied before the provers are called.

Carrying out interactive proofs using characteristic formulae certainly also requires some expertise with the interactive theorem prover and its proof-search features. However, interactive proofs give near-instantaneous feedback, saving overall a lot of time in the verification process. In the end, an interactive proof script might

involve more lines than the integrated proof hints of Jahob, however those lines might take a lot less time to come up with.

Pangolin Pangolin, developed by Régis-Gianas and Pottier [76] is the first VCG tool that targets a higher-order, purely-functional programming language. The key innovation of this work is the treatment of first-class functions: a function gets reflected in the logic as the pair of its pre-condition and of its post-condition. More recently, the tool Who [44], by Kanig and Filliâtre, builds upon the same idea for extending Why [26] with imperative higher-order functions. I have followed a very different approach to lifting functions to the logical level. In my work, functions are represented as values of the abstract data type Func and specified using the abstract predicate AppReturns. This approach has three benefits compared with representing functions as pairs of a pre- and a post-condition.

Firstly, the predicate AppReturns allows assigning several specifications to the same functions, whereas in Pangolin the specification of a function is hooked into the type of the function. Assigning multiple specifications to a same function turns out to be particularly useful for higher-order functions, which typically admit a complex most-general specification and a series of simpler, specialized specifications.

Secondly, the fact that functions are uniformly represented as a data type Func rather than as a pair of a pre- and a post-condition makes it simpler to reason about functions stored in data-structures. For example, in my approach a list of functions is reflected as the type "List Func" while in Pangolin the same list would have a type of the form "List $(\exists AB.\ (A \to \mathsf{Prop}) \times (B \to \mathsf{Prop}))$", involving dependent types.

Thirdly, specification of functions through the predicate AppReturns interacts better with ghost variables, as the reflection of a function f as a logical value of type "$(A \to \mathsf{Prop}) \times (B \to \mathsf{Prop})$" does not take ghost variables into account. Although Pangolin offers syntactic sugar for specifying functions with ghost variables, this syntactic sugar has to be ultimately eliminated, leading to duplication of hypotheses in pre- and post-conditions or in proof obligations.

8.4 Shallow embeddings

Program verification with a shallow embedding consists in relating the source code of the program that is executed with a program written in the logic of a theorem prover. This approach can take several forms.

Programming in type theory with ML types Leroy's formally-verified C compiler [47] is, for the most part, programmed directly in Coq. The extraction mechanism of Coq is used to obtain OCaml source code out of the Coq definitions. This translation is mainly a matter of adapting the syntax. Because the logic of Coq includes only pure total functions, the compiler is implemented without any imperative data structure, and the termination of all the recursive functions involved is justified through a simple syntactic criteria. Although those restrictions appeared

acceptable for writing a program such as a compiler, there are many programs that cannot accommodate such restrictions.

The source code of Leroy's compiler involves only basic ML types. Typically, a data structure is represented using algebraic types, and the properties of a value of type T are specified through a predicate of type $T \to \mathsf{Prop}$. There exists another style of programming, detailed next, which consists in using dependent types to enforce invariants of values, for example relying on the type "$\mathsf{list}\, n$" to describe lists of length n.

Programming in type theory with dependent types Programming with dependent types has been investigated in particular in Epigram [54], Adga [19] and Russell [81]. The latter is an extension to Coq, which behaves as a permissive source language which elaborates into Coq terms. In Russell, establishing invariants, justifying termination of recursive functions and proving the inaccessibility of certain branches from a pattern matching can be done through interactive Coq proofs. Those features make it easier in particular to write programs featuring nontrivial recursion (even though the proofs about such dependently-typed functions might turn out to be more technical).

When dependent types are used, Coq functions manipulate values and proofs in the same time. So, the work of the extraction mechanism is more involved than when programming with ML types, because the proof-specific entities need to be separated from the values with computational content. Recognizing proof-specific elements is far from trivial. With the current implementation of Coq, some ghost variables might fail to be recognized as computationally irrelevant. Such variables remain in the extracted code, leading to runtime inefficiencies and, possibly to incorrect asymptotic complexity [81]. The Implicit Calculus of Constructions [6] could solve that problem, as it allows tagging ghost variables manually, however this system is not yet implemented.

Hoare Type Theory and Ynot The two approaches described so far do not support imperative features. The purpose of Hoare Type Theory (HTT) [61, 63], developed by Nanevski, Morrisett and Birkedal, is precisely to extend the programming language from the logic with an axiomatic monad for describing impure features such as side-effects and non-termination. HTT is implemented in a tool called Ynot [17] and it has been recently re-implemented by Nanevksi *et al* [64]. Both developments, implemented in Coq, allow for the extraction of code that performs side effects. The type of the monad involved takes the form "$\mathsf{ST}\, P\, Q$", which is a partial-correctness Hoare triple asserting that a term of that type admits the pre-condition P and the post-condition Q. On top of the monadic type ST is built a Separation Logic-style monad, of type "$\mathsf{STsep}\, P\, Q$", which supports local reasoning.

In HTT, verification proofs thus take the form of Coq typing derivations for the source code. So, program verification is done at the same time as writing 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 can 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, because pattern matching is so deeply hard-wired in Coq that it is not straightforward to extend it.

Another technical difficulty faced by HTT is the treatment of ghost variables. A specification of the form "$\mathsf{ST}\,P\,Q$" does not naturally allow for ghost variables to be used for sharing information between the pre- and the post-condition. Indeed, if P and Q both refer to a ghost variable x quantified outside of the type "$\mathsf{ST}\,P\,Q$", then x is considered as a computationally-relevant value and thus it appears in the extracted code (indeed, x is typically not of type Prop). Ynot [17] relies on a hack for simulating the Implicit Calculus of Constructions [6], which, as mentioned earlier on, allows to tag ghost variables explicitly. A danger of this approach is that forgetting to tag a variable as ghost does not produce any warning yet results in the extracted code being inefficient. HTT [63, 64] takes a different approach and implements post-conditions as predicate over both the input heap and the output heap. This makes it possible to eliminate ghost variables by duplicating the pre-condition inside the post-condition. Typically, "$\forall x.\ \mathsf{ST}\,P\,Q$" gets encoded as "$\mathsf{ST}\,(\exists x.\,P)\,(\forall x.\,P \Rightarrow Q)$". Tactics are then developed to avoid the duplication of proof obligations, however the duplication remains visible in specifications. It might be possible to also hide the duplication in specifications with a layer of notation, however such a notation system does not appear to have been implemented.

Translation of code into logical definitions The extraction mechanism involved in the aforementioned approaches take a program described as a logical definition and translate it into a conventional programming language. It is also possible to consider a translation that goes instead in the other direction, from a piece of code towards a higher-order logic definition.

The LOOP compiler [43] takes Java programs and compiles them into PVS definitions. It supports a fairly large fragment of the Java language and features advanced PVS tactics for reasoning on generated definitions. In particular, it includes a weakest-precondition calculus mechanism that allows achieving a high degree of automation, despite being limited in the size of the fragment of code that can be automatically verified. However, interactive proofs require a lot of expertise from the user: LOOP requires not only to master the PVS tool but also to understand the compilation scheme involved [43]. By contrast, the tactics manipulating characteristic formulae appear to allow conducting interactive proofs of correctness without detailed knowledge on the construction of those formulae.

Myreen and Gordon [60, 58] decompile machine code into HOL4 functions. The lemmas proved interactively about the generated HOL4 functions can then be automatically transformed into lemmas about the behavior of the corresponding pieces of machine code. This approach has been applied to verify a complete LISP inter-

preter implemented in machine code. The translation into HOL4 is possible only because the functional translation of a while loop is a tail-recursive function. Indeed, 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 [50]. 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.

SeL4 The goal of the seL4 project [46, 45] is the machine-checked verification of the seL4 microkernel. The microkernel is implemented in a subset of C, and also includes lines of assembly code. The development involves two layers of specification. First, abstract logical specifications, describing the high-level invariants that the microkernel should satisfy, have been stated in Isabelle/HOL. Second, a model of the microkernel has been implemented in a subset of Haskell. This executable specification is automatically translated into Isabelle/HOL definitions, via a shallow embedding.

The proof of correcteness is two-step. First, it involves relating the translation of the executable Haskell specification with a deep embedding of the C code. This task can be partially automated with the help of specialized tactics. Second, the proof involves relating the abstract specification with the translation of the executable specification. This task is mostly carried out at the logical level, and thus does not involve referring to the low-level representation of data-structures such as doubly-linked lists.

To summarize, this two-step approach involves a shallow embedding of some source code. This code is not the low-level C code that is ultimately compiled, but rather the source code of an abstract version of that code, expressed in a higher-level programming language, namely Haskell. The shallow embedding approach applies well in seL4 because the code of the executable specification was written in such a way as to avoid any nontrivial recursion [45]. Overall, the approach is fairly similar to that of Myreen and Gordon [60, 58], except that the low-level code is not decompiled automatically but instead decompiled by hand, and this decompilation is proved correct using semi-automated tactics.

The KeY system The KeY system [7] is a fairly successful approach to the verification of Java programs. Contrary to the aforementioned approaches, the KeY system does not target a standard mathematical logic, but instead relies on a logic specialized for reasoning on imperative programs, called Dynamic Logic. Dynamic Logic [33] is a particular kind of modal logic in which program code appears inside modalities such as the *mix-fix* operator $\langle\cdot\rangle$. For example, "$H_1 \to \langle t \rangle\, H_2$" is a Dynamic Logic formula asserting that, in any heap satisfying H_1, the sequence of commands t terminates and produces a heap satisfying H_2. This formula thus has the same interpretation as the proposition "$[\![t]\!]\, H_1\, (\#\, H_2)$". Reasoning rules of Dynamic Logic

enable symbolic execution of source code that appears inside a modality. Reasoning on loops and recursive functions can be conducted by induction. Local reasoning is supported in the KeY methodology thanks to the recent addition of a feature called Dynamic Frames [79].

Despite being based on very different logics, the characteristic formulae approach and the KeY approach to program verification show a lot of similarities regarding the high-level interactions between the user and the system. Indeed, KeY features a GUI interface where the user can view the current formula and make progress by writing a proof script made of tactics (called "taclets"). It also relies on existential variables for delaying the instantiations of particular intermediate invariants. Thanks to those similarities, the approach based on characteristic formulae can presumably leverage on the experience acquired through the KeY project and take inspiration from the nice features developed for the KeY system. In the same time, by being implemented in terms of a mainstream theorem prover rather than a custom one, characteristic formulae can leverage on the fast development of that theorem prover and benefit from the mathematical libraries that are developed for it.

8.5 Deep embeddings

A fourth approach to formally reasoning on programs consists in describing the syntax and the semantics of a programming language in the logic of a proof assistant, using inductive definitions. In theory, the deep embedding approach can be used to verify programs written in any programming language, and without any restriction in terms of expressiveness (apart from those of the proof assistant).

Proof of concept by Mehta and Nipkow Mehta and Nipkow [56] have set up the first proof-of-concept of program verification through a deep embedding in a general-purpose theorem prover. The authors axiomatized the syntax and the semantics of a basic procedural language in Isabelle/HOL. Then, they proved Hoare-style reasoning rules correct with respect to the semantics of that language. Finally, they implemented in ML a VCG tactic for automatically applying the reasoning rules. The VCG tactic, when applied to a source code annotated with specification and invariants, produces a set of proof obligations that entails the correctness of the source code. The approach was validated through the verification of a short yet complex program, the Schorr-Waite graph-marking algorithm. Although Mehta and Nipkow's work is conducted inside a theorem prover, it remains fairly close to the traditional VCG-based approach.

To reason about data that are chained in the heap, Mehta and Nipkow adapted a technique from Bornat [12]. They defined a predicate "$\mathsf{List}\, m\, l\, L$" to express the fact that, in a memory store m, there exists a path that goes from the location l to the value null by traversing the locations mentioned in the list L. Mehta and Nipkow observed that the definition of the predicate List allows for some form of local reasoning, in the sense that the proposition "$\mathsf{List}\, m\, l\, L$" remains true when

modifying a pointer that does not belong to the list L. However, the predicate List refers to the entire heap m, so it does not support application of the frame rule.

The frameworks XCAP and SCAP The frameworks XCAP [65] and SCAP [25], developed by Shao *et al*, rely on deep embeddings for reasoning in Coq about assembly programs. They support reasoning on advanced patterns such as strong updates, embedded code pointers, or longjump/setjump. Those frameworks have been used to verify short but complex assembly routines, whose proof involves hundreds of lines per instruction.

One of the long-term goals of the Flint project headed by Shao is to develop the techniques required to formally prove correct the kernel of an operating system. Such a development involves reasoning at different abstraction levels. Typically, a thread scheduler or a garbage collector needs to have a lower-level view of memory, whereas threads that are executed by the scheduler and that use the garbage collection facility might have a much higher-level view of memory. So, it makes sense to develop various logics for reasoning at each abstraction layer. Yet, at some point, all those logics must be related to each other.

As Shao *et al* have argued, the key benefits of using a deep embedding is that it allows for a foundational approach to program verification, in which all those logics are ultimately defined in terms of the semantics of machine code. In this thesis, I have focused on reasoning on programs written in a high-level programming language, so the challenge is quite different. Applying such a foundational approach to characteristic formulae would involve establishing a formal connection between a characteristic formula generator and a formally-verified compiler.

From a deep embedding of pure-Caml to characteristic formulae The development of the characteristic formulae presented in this thesis originates in an investigation of the use of a deep embedding for proving purely-functional programs. In the development of this deep embedding, I did formalize in Coq the syntax and the semantics of the pure fragment of Caml. The specification predicate took the form "$t \Downarrow\!| P$", asserting that the execution of the term t terminates and returns a value satisfying the predicate P. The reasoning rules, which take the form of lemmas proved correct with respect to the axiomatized semantics, were used to establish judgments of the form "$t \Downarrow | P$".

I relied on tactics to help applying those reasoning rules. Those tactics behaved in a similar way as the `x`-tactics developed for characteristic formulae, although the implementation of those tactics was much more involved (I explain why further on). With this deep embedding, I was able to verify the implementation of the list library of OCaml and to prove correct a bytecode compiler and interpreter for mini-ML. The resulting verification proof scripts were actually quite similar to those involved when working with characteristic formulae. Further details can be found in the corresponding technical report [15]. I next explain how the deep embedding reasoning rules led to characteristic formulae, an what improvements were brought

by characteristic formulae.

When verifying a program through a deep embedding, the reasoning rule are applied in a very systematic manner. For example, whenever reaching a let-node, the reasoning rules for let-bindings, shown next, needs to be applied.

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

The intermediate specification P', which does not appear in the goal, typically needs to be provided at the time of applying the rule. The introduction of an existential quantifier allows applying the rule by delaying the instantiation of P'. The updated reasoning rule for let-bindings is as follows.

$$\big(\exists P'.\ (t_1 \Downarrow P') \wedge (\forall x.\ P'\,x \Rightarrow t_2 \Downarrow P)\big) \quad \Rightarrow \quad ((\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2) \Downarrow P)$$

This new reasoning rule is now entirely goal-directed, so it can be applied in a systematic manner, following the source code of the program. This led to the characteristic formula for let-bindings, recalled next. Observe that "$[\![t]\!]\,P$" has exactly the same interpretation as "$t \Downarrow P$", since both assert that the term t returns a value satisfying P.

$$[\![\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2]\!]\,P \quad \equiv \quad \big(\exists P'.\ [\![t_1]\!]\,P' \ \wedge\ (\forall x.\,P'\,x \Rightarrow [\![t_2]\!]\,P)\big)$$

In addition to automating the application of reasoning rules, another major improvement brought by characteristic formulae concerns the treatment of program values. With characteristic formulae, a list of integers "$3 :: 2 :: \mathsf{nil}$" occurring in a Caml program is described as the corresponding Coq list of integers. However, with the deep embedding, the same value is represented as:

$$\mathsf{vconstr}_2\ \mathsf{cons}\ (\mathsf{vint}\,3)\ (\mathsf{vconstr}_2\ \mathsf{cons}\ (\mathsf{vint}\,2)\ (\mathsf{vconstr}_0\ \mathsf{nil}))$$

where vconstr is the constructor from the grammar of values used to represent the application of data constructors (the index that annotates vconstr indicates the arity of the data constructor), and where vint is the constructor from the grammar of values used to represent program integers. I did develop an effective technique for avoiding to pollute specifications and reasoning with such a low-level representation. For every Coq type A, I generated the Coq definition of its associated encoder, which is a total function of type $A \to \mathsf{Val}$, where Val denotes the type of Caml values in the deep embedding. The actual specification judgment in the deep embedding took the form "$t \Downarrow A\,|\,P$", where A is a type for which there exists an encoder of type "$A \to \mathsf{Val}$", and where P is a predicate of type "$A \to \mathsf{Prop}$". The proposition "$t \Downarrow A\,|\,P$" asserts that the evaluation of the term t returns the encoding of a value V of type A such that $P\,V$ holds. With this predicate, program verification through a deep embedding could be conducted almost entirely at the logical level.

From my experience of developing a framework both for a deep embedding and for characteristic formulae, I conclude that moving to characteristic formulae brings at least three major improvements. First, characteristic formulae do not need to

represent and manipulate program syntax. Thus, they avoid many technical difficulties, in particular those associated with the representation of binders. Moreover, I observed that the repeated computations of substitutions that occur during the verification of a deeply-embedded program can lead to the generation of proof terms of quadratic size, which can be problematic for scaling up to larger programs.

Second, with characteristic formulae there is no need to apply reasoning rules of the program logic, because the applications of those rules is automatically done through the construction of the characteristic formulae. In particular, characteristic formulae saves the need to compute reduction contexts. With a deep embedding, to prove a goal of the form "$t \Downarrow | P$", one must first rewrite the goal in the form "$C[t_1] \Downarrow | P$", where C is a reduction context and where t_1 is the subterm of t in evaluation position. More generally, by moving to characteristic formulae, tactics could be given a much simpler and a much more efficient implementation.

Third and last, characteristic formulae, by lifting values at the logical level once and for all at the time of their construction, completely avoid the need to refer to the relationship between program values and logical values. Contrary to the deep embedding approach, characteristic formulae do not require encoders to be involved in specifications and proofs; encoders are only involved in the proof of soundness of characteristic formulae. The removal of encoders leads to simpler specifications, simpler reasoning rules, and simpler tactics.

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

Chapter 9

Conclusion

This last chapter begins with a summary of how the various approaches to program verification try to avoid referring to program syntax. I then recall the main ingredients involved for constructing a practical verification tool upon the idea of characteristic formulae. Finally, I discuss directions for future research, and conclude.

9.1 Characteristic formulae in the design space

The correctness of a program ultimately refers to the semantics of the programming language in which that program is written. The reduction judgment describing the semantics, written $t_{/m} \Downarrow v'_{/m'}$ in the thesis, directly refers to the input memory state m and to the output memory state m'. Hoare-triples, here written $\{H\}\ t\ \{Q\}$, allow for abstraction in specifications. While a memory state m describes exactly what values lie in memory, the heap descriptions involved in Hoare triples allow to state properties about the values stored in memory, without necessarily revealing the exact representation of those values. Separation Logic later suggested an enhanced interpretation of Hoare triples, according to which any allocated memory cell that is not mentioned in the pre-condition of a term is automatically guaranteed to remain unchanged through the evaluation of that term. Separation Logic thereby allows for local reasoning, in the sense that the verification of a program component can be conducted independently of the pieces of state owned by other components.

By moving from a judgment of the form $t_{/m} \Downarrow v'_{/m'}$ to a judgment of the form $\{H\}\ t\ \{Q\}$, Hoare logic and Separation Logic avoid a direct reference to the memory states m and m'. However, the reference to the source code t still remains. In the traditional VCG approach, this is not an issue: when the code is annotated with sufficiently-many specifications and invariants, it is possible to automatically extract a set of verification conditions. Proving the verification condition then ensures that the program satisfies its specification. Although the VCG approach works smoothly when the verification conditions can be discharged by fully-automated theorem provers, verification conditions are not as well-suited for interactive proofs.

One alternative to building Hoare-logic derivations automatically consists in building them by hand, through interactive proofs. To build the derivation of a Hoare triple $\{H\}\ t\ \{Q\}$ manually, one has to refer to the source code t explicitly. The description of source code in the logic involves a deep embedding of the programming language. Program verification through a deep embedding is possible, however it requires a significant effort, mainly because values from the programming language needs to be explicitly lifted at the logical level, and because reasoning on reduction steps involves computing substitutions in the deep embedding of the source code.

The main motivation for verifying programs with a shallow embedding is precisely to avoid the reference to programming language syntax. There are several ways in which a shallow embedding can be exploited in program verification, but they all rely on the same idea: building a logical definition that corresponds to the source code of the program to be verified. These techniques relies on the fact that the logic of a theorem prover actually contains a small programming language. Yet, the shallow embedding approach suffers from restrictions related to the fact that some features of the host logical language are carved in stone.

I have followed a different approach to removing direct reference to the source code of the program to be verified. It consists in generating a logical formula that characterizes the behavior of the source code. This concept of characteristic formulae is not new: it was first developed in process calculi [70, 57] and was later adapted to PCF programs by Honda, Berger and Yoshida [40]. Yet, those characteristic formulae for PCF programs work did not lead to a practical verification system, because the specification language relied on a nonstandard logic.

In this thesis, I have shown how to construct characteristic formulae expressed in a standard higher-order logic. The characteristic formula of a term t is a predicate, written $[\![t]\!]$, such that $[\![t]\!]\,H\,Q$ implies that the term t admits H as pre-condition and Q as post-condition. I have implemented a tool for generating Coq characteristic formulae for Caml programs, and I have developed a Coq library that includes definitions, lemmas and tactics for proving programs through their characteristic formula. Finally, I have applied this tool for specifying and verifying nontrivial programs. The key ingredients involved for turning the idea of characteristic formulae into a practical verification system are summarized next.

9.2 Summary of the key ingredients

Characteristic formulae The starting point of this thesis is the notion of characteristic formula, which is a higher-order logic formulae that characterizes the set of total-correctness specifications that a given program admits. A characteristic formula is built compositionally from the source code of the program it describes, and it has a size linear in that of the program. The program does not need to be annotated with specification and invariants, as unknown specifications and invariants get quantified existentially in characteristic formulae.

Notation layer If $\mathcal{F}$ is the characteristic formula of a term t, the proof obligation takes the form "$\mathcal{F}\,H\,Q$", where H is the pre-condition and Q is the post-condition. I have devised a system of notation for pretty-printing characteristic formulae in a way that closely resemble the source code that the formula describes. So, the proof obligation "$\mathcal{F}\,H\,Q$" reads like the term t being applied to the pre- and the post-conditions, in the form "$t\,H\,Q$". Since characteristic formulae are built compositionally, the ability to easily read proof obligations applies not only to the initial term t but also to all of its subterms.

Reflection of values The values involved in the execution of a program are specified using values from the logic. For example, if a Caml function expects a list of integers, then its pre-condition is a predicate about Coq list of integers. This reflection of Caml values as Coq values works for all values except for functions, because Coq functions can only describe total functions. I have introduced the abstract type Func to represent functions, as well as the abstract predicate AppEval for specifying the behavior of function applications. In the soundness proof, a value of type Func is interpreted as the deep embedding of the source code of some function. This interpretation avoids the need to rely on a nonstandard logic, as done in the work of Honda, Berger and Yoshida [40].

Ghost variables The specification of a function generally involves ghost variables, in particular for sharing information between the pre-condition and the post-condition. In order to state general lemmas about function specifications, such as the induction lemma which inserts an induction hypothesis into given specification, I have introduced the predicate Spec. The proposition SpecfK asserts that the function f admits the specification K, where K is an higher-order predicate able to capture the quantification over an arbitrary number of ghost variables.

N-ary functions Caml programs typically define functions of several arguments in a curried fashion. Partial applications are commonplace, and over-applications are also possible. In theory, a function of two arguments can always be specified as a function of one argument that returns another function of one argument. Yet, a higher-level predicate that captures the specification of a function of two arguments in a more direct way is a must-have for practical verification. For that purpose, I have defined a family of predicates AppReturns$n$ and Spec$_n$, which are all ultimately defined in terms of the core predicate AppReturns.

Heap predicates Characteristic formulae, which were first set up for purely functional programs, could easily be extended to an imperative setting by adding heap descriptions. I exploited the techniques of Separation Logic for describing heaps in a concise way, and followed the Coq definitions of Separation Logic connectives developed in Ynot [17]. In particular, heap predicates of the form $L \hookrightarrow_{\mathcal{T}} V$ support reasoning on strong updates. One novel ingredient for handling Separation Logic in

characteristic formulae is the definition of the predicate local, which is inserted at every node of a characteristic formula, thereby allowing for application of the frame rule at any time.

Bounded recursive ownership The use of representation predicates is a standard technique for relating imperative data structures with their mathematical model. I have extended this technique to polymorphic representation predicates: a representation predicate for a container takes as argument the representation predicate describing the elements stored in that container. Depending on the representation predicate given for the elements, one can describe either a situation where the container owns its elements, or a situation where the container does not own its elements and simply views them as base values.

Weak-ML ML types greatly help in program verification because they allow reflecting basic Caml values directly as their Coq counterpart. Yet, for the sake of type soundness, the ML type system keeps track of the type of functions, and it enforces the invariant that a location of type $\mathsf{ref}\,\tau$ contains a value of type τ at any time. Since program verification is a much more accurate analysis than ML type-checking, it need not suffer from the restrictions imposed by ML. The introduction of the type system weak-ML serves two purposes. First, it relaxes the restrictions associated with ML, in particular through a totally-unconstrained typing rule for applications. Second, it simplifies the proof of soundness of characteristic formulae, by allowing to establish a bijection between well-typed weak-ML values and a subset of Coq values, and by avoiding the need to involve a store typing oracle.

High-level tactics I have developed a set of tactics for reasoning on characteristic formulae in Coq. Those tactics not only shorten proof scripts, they also make it possible to verify programs without knowledge of the details of the construction of characteristic formulae. The end-user only needs to learn the various syntaxes for CFML tactics, in particular the optional arguments for specifying how variables and hypotheses should be named. Furthermore, in the verification of imperative programs, tactics play a key role by supporting application of lemmas modulo associativity and commutativity of the separating conjunction, and by automating the application of the frame rule on function calls.

Implementation The CFML generator parses Caml code and produces Coq definitions. It reuses the parser and the type-checker of the OCaml compiler, and involves about 3,000 lines of OCaml for constructing characteristic formulae. The CFML library contains notations, lemmas and tactics for manipulating characteristic formulae. It is made of about 4,000 lines of Coq. Note that the CFML library would have been significantly easier to implement if Coq did feature a slightly more evolved tactic language.

9.3 Future work

Formal verification of characteristic formulae Defenders of a foundational approach to program verification argue that we ought to try and reduce the trusted computing base as much as possible. There are at least two ways of proceeding. The first one involves building a verified characteristic formula generator and then conducting a machine-checked proof of the soundness theorem for characteristic formulae. An alternative approach consists in developing a program that, instead of generating axioms to reflect every top-level declaration from the source Caml program, directly generates lemmas and their proofs. Those proofs would be carried out in terms of a deep embedding of the source program. Note that the program generating the lemmas and their proofs would not need to be itself proved correct, because its output would be validated by the Coq proof-checker. I expect both of those approaches to involve a lot of effort if a full-blown programming language is to be supported.

Non-determinism In this thesis, I have assumed the source language to be deterministic. This assumption makes definitions and proofs slightly simpler, in particular for the treatment of partial functions and polymorphism, however it does not seem to be essential to the development. One could probably replace the predicate $\mathsf{AppEval}$ with two predicates: a deterministic predicate describing only the evaluation of partial applications of curried functions, and a non-deterministic predicate describing the evaluation of general function applications.

Exceptions It should be straightforward to add support for throwing and catching of exceptions. Currently, a post-condition has the type "$A \to \mathsf{Hprop}$", describing both the result value of type A and the heap produced. Support for exceptions can be added by generalizing the result type to "$(A + \mathsf{exn}) \to \mathsf{Hprop}$", where exn denotes the type of exceptions. Such a treatment of exceptions has been implemented for example in Ynot [17]. Characteristic formulae would then involve a left injection to describe a normal result and involve a right injection to describe the throwing of an exception.

Arithmetic So far, I have completely avoided the difficulties related to fixed-size representations of numbers: I have assumed arbitrary-precision integers and I have not provided any support for floating-point numbers. One way to obtain correct programs without wasting time reasoning on modulo arithmetic consists in representing all integers using the BigInt library. Alternatively, one could target fixed-size integers, yet at the cost of proving side-conditions for every arithmetic operation involved in the source program. For floating-point numbers, the problem appears to be harder. Nevertheless, it should be possible to build upon the recent breakthroughs in formal reasoning on numerical routines [11, 3] and achieve a high degree of automation.

Fields as arrays The frame rule can be made even more useful if the ownership of an array of records can be viewed as the ownership of a record of arrays. Such a re-organization of memory allows to invoke the frame rule on particular fields of a record. One can then obtain for free the property that a function that needs only access one field of a record does not modify the other fields. It should be possible to define heap predicates that capture the field-as-array methodology.

Hidden state Modules can be used to abstract the definition of the invariants of private pieces of heap. One may want to go further and hide altogether the existence of such a private piece of state. This is precisely the purpose of the anti-frame rule developed by Pottier [75, 80]. For example, assume that a module exports a heap predicate I of type Hprop and a function f that admits the pre-condition $H * I$ and the post-condition $Q \star I$. Assume that the module also contains a private initialization code that allocates a piece of heap satisfying the invariant I. Then, the module can be viewed from the outside world as simply exporting a function f that admits the pre-condition H and the post-condition Q, removing any reference to the invariant I. Integrating the anti-frame rule would allow exporting cleaner specifications for modules such as a random-number generator, a hash-consing data structure, a memoization module, and, more generally, for any imperative data structure that exposes a purely-functional interface.

Concurrency The verification of concurrent programs is a obvious direction for future research. A natural starting point would be to try and extend characteristic formulae with the reasoning rules developed for Concurrent Separation Logic, which was initially devised by O'Hearn and Brookes [13, 68], and later also developed by other researchers [72, 37].

Module systems In the experimental support for modules and systems, I have mapped Caml modules to Coq modules. However, limitations in the current module system of Coq make it difficult to verify nontrivial Caml functors. The enhancement of modules for Coq is a largely-open problem. In fact, one may argue that even the module system of Caml is not entirely satisfying. I hope that, in the near future, an expressive and practical module system that fits both Caml and Coq will be proposed. Such a common module system would be very helpful for carrying out modular verification of modular programs without having to hack around several limitations.

Low-level languages It would be very interesting to implement a characteristic formula generator for a lower-level programming language such as C or assembly. Features such as pointer arithmetic should not be too difficult to handle. The treatment of general "goto" instructions may be tricky to handle, however the treatment of "break" and "continue" instructions appears to be straightforward to integrate

in the characteristic formulae for loops. Characteristic formulae for low-level programs should also bring strong guarantees about total collection, ensuring that all the memory allocated by a program is freed at some point.

Object-oriented languages It would also be worth investigating how characteristic formulae may be adapted to an object-oriented programming language like Java or C#. The object presentation naturally encourages ghost variables, which describes the mathematical model of an object, to be laid out as ghost fields. This treatment may reduce the number of existentially-quantified variables and thereby allow more goals to be solved by automated theorem provers. Yet, a number of difficult problems arise when moving to an object-oriented language, in particular with respect to the specification of collaborating classes and thus need to share a common invariant, and with respect to the inheritance of specifications. It would be interesting to see whether characteristic formulae can integrate the notion of dynamic specification that has been developed by Parkinson and Bierman [71] to handle inheritance in Java.

Partial correctness The characteristic formulae that I have developed target total correctness, which offers a stronger guarantee than partial correctness. Yet, some programs, such as web-servers, are not intended to terminate. A partial-correctness Hoare triple whose post-condition is the predicate False asserts that a program diverges, because the program is proved never to crash and never to terminate. Divergence cannot be specified in a total correctness setting. So, one could be interested in trying to build characteristic formulae that target partial correctness, possibly reusing ideas developed by Honda *et al* [40].

Lazy evaluation The treatment of lazy evaluation that I relied on for verifying Okasaki's data structures is quite limited. Indeed, verifying a program by removing all the lazy keywords for the source code is sound but not complete. I have started to work on an explicit treatment of suspended computations, aiming for a proper model of the implementation of laziness in Caml. More challenging would be the reasoning on Haskell programs, where laziness is implicit. The specification of Haskell programs would probably involve the specification of partially-defined data structures, as suggested by Danielsson *et al* [20].

Complexity analysis One could generalize characteristic formulae into predicates that, in addition to a pre- and a post-condition, also applies to a bound on the number of reduction steps. Proof obligations would then take the form "$\mathcal{F}\,N\,H\,Q$", where $\mathcal{F}$ is a characteristic formula and N is a natural number. Another possible approach, suggested by Pottier, consists in keeping proof obligations in the form "$\mathcal{F}\,H\,Q$" and adding an artificial heap predicate of the form "$\mathsf{credit}\,N$", which denotes the ownership of "the right to perform N reduction steps". Note that a credit of the form "$\mathsf{credit}\,(N+M)$" can be split at any time into a conjunction "$\mathsf{credit}\,N * \mathsf{credit}\,M$".

This suggestion has been studied by Pilkiewicz and Pottier [74] in a type system that extends the type system developed by Pottier and myself [16]. It should not be difficult to modify characteristic formulae so as to impose that atomic operations consume exactly one credit, in the sense that they expect "credit 1" in the pre-condition and do not give this credit back in the post-condition. Combined with a treatment of laziness, this approach based on credits would presumably enable the formalization of the advanced complexity analyses involved in Okasaki's book [69].

9.4 Final words

I would like to conclude by speculating on the role of interactive theorem provers in future developments of program verification. Research on artificial intelligence has had limited success in automated theorem proving, and, in spite of the impressive progress made by SMT solvers, the verification of nontrivial programs still requires a good amount of user intervention in addition to the writing of specifications. Being established that some form of interaction between the user and the machine is required, proof assistants appear as a fairly effective way for users to provide the high-level arguments of a proof and for the system to give real-time feedback. My thesis has focused on the development of characteristic formulae, which aim at smoothly integrating program verification within proof assistants. Proof assistants have undergone tremendous improvements in the last decade. Undoubtedly, they will continue to be improved, making interactive proofs, and, in particular, program verification, more largely applicable and more accessible.

Bibliography

[1] L. Aceto and A. Ingolfsdottir. Characteristic formulae: from automata to logic. *Bulletin of the European Association for Theoretical Computer Science*, 91:57, 2007. Columns: Concurrency.

[2] Andrew W. Appel. Tactics for separation logic. *Unpublished draft, http://www.cs.princeton.edu/appel/papers/septacs.pdf*, 2006.

[3] Ali Ayad and Claude Marché. Multi-prover verification of floating-point programs. In Jürgen Giesl and Reiner Hähnle, editors, *International Joint Conference on Automated Reasoning*, Lecture Notes in Artificial Intelligence, Edinburgh, Scotland, 2010. Springer.

[4] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects (FMCO) 2005, Revised Lectures*, volume 4111 of *LNCS*, pages 364–387, New York, NY, 2006. Springer-Verlag.

[5] 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.

[6] Bruno Barras and Bruno Bernardo. The implicit calculus of constructions as a programming language with dependent types. In Roberto M. Amadio, editor, *FoSSaCS*, volume 4962 of *LNCS*, pages 365–379. Springer, 2008.

[7] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer-Verlag, Berlin, 2007.

[8] 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.

[9] Josh Berdine and Peter W. O'Hearn. Strong update, disposal, and encapsulation in bunched typing. In *Mathematical Foundations of Programming Seman-*

tics, volume 158 of *Electronic Notes in Theoretical Computer Science*, pages 81–98. Elsevier Science, 2006.

[10] Martin Berger, Kohei Honda, and Nobuko Yoshida. A logical analysis of aliasing in imperative higher-order functions. In *ACM International Conference on Functional Programming (ICFP)*, pages 280–293, 2005.

[11] Sylvie Boldo and Jean-Christophe Filliâtre. Formal verification of floating-point programs. In *IEEE Symposium on Computer Arithmetic*, pages 187–194. IEEE Computer Society, 2007.

[12] Richard Bornat. Proving pointer programs in Hoare logic. In *International Conference on Mathematics of Program Construction (MPC)*, volume 1837 of *LNCS*, pages 102–126. Springer, 2000.

[13] Stephen D. Brookes. A semantics for concurrent separation logic. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR*, volume 3170 of *LNCS*, pages 16–34. Springer, 2004.

[14] R. M. Burstall. Some techniques for proving correctness of programs which alter data structures. In B. Meltzer and D. Mitchie, editors, *Machine Intelligence 7*, pages 23–50. Edinburgh University Press, Edinburgh, Scotland., 1972.

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

[16] Arthur Charguéraud and François Pottier. Functional translation of a calculus of capabilities. In *ACM International Conference on Functional Programming (ICFP)*, 2008.

[17] Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective interactive proofs for higher-order imperative programs. In *ACM International Conference on Functional Programming (ICFP)*, 2009.

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

[19] Thierry Coquand. Alfa/agda. In Freek Wiedijk, editor, *The Seventeen Provers of the World, Foreword by Dana S. Scott*, volume 3600 of *LNCS*, pages 50–54. Springer, 2006.

[20] Nils Anders Danielsson, John Hughes, Patrik Jansson, and Jeremy Gibbons. Fast and loose reasoning is morally correct. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 206–217, 2006.

[21] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis (TACAS)*, volume 4963 of *LNCS*, pages 337–340, Berlin, 2008. Springer-Verlag.

[22] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[23] Manuel Fähndrich and Robert DeLine. Adoption and focus: practical linear types for imperative programming. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 13–24, 2002.

[24] Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In Rocco De Nicola, editor, *ESOP*, volume 4421 of *LNCS*, pages 173–188. Springer, 2007.

[25] Xinyu Feng, Zhong Shao, Alexander Vaynberg, Sen Xiang, and Zhaozhong Ni. Modular verification of assembly code with stack-based control abstractions. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 401–414, 2006.

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

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

[28] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 237–247, 1993.

[29] R. W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, 1967.

[30] Georges Gonthier and Assia Mahboubi. A small scale reflection extension for the Coq system. Research Report 6455, Institut National de Recherche en Informatique et en Automatique, Le Chesnay, France, 2008.

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

[32] 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.

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

[34] Eric C. R. Hehner. Specified blocks. In Bertrand Meyer and Jim Woodcock, editors, *VSTTE*, volume 4171 of *LNCS*, pages 384–391. Springer, 2005.

[35] M. C. B. Hennesy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, 1985.

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

[37] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle semantics for concurrent separation logic. In Sophia Drossopoulou, editor, *ESOP*, volume 4960 of *LNCS*, pages 353–367. Springer, 2008.

[38] Kohei Honda. From process logic to program logic. In Chris Okasaki and Kathleen Fisher, editors, *ICFP*, pages 163–174. ACM, 2004.

[39] Kohei Honda, Martin Berger, and Nobuko Yoshida. An observationally complete program logic for imperative higher-order functions. In *Proceedings of the Twentieth Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 280–293, 2005.

[40] Kohei Honda, Martin Berger, and Nobuko Yoshida. Descriptive and relative completeness of logics for higher-order functions. In M. Bugliesi, B. Preneel, V. Sassone, and I. Wegener, editors, *Automata, Languages and Programming, 33rd International Colloquium, ICALP 2006, Venice, Italy, July 10-14, 2006, Proceedings, Part II*, volume 4052 of *LNCS*. Springer, 2006.

[41] Kohei Honda and Nobuko Yoshida. A compositional logic for polymorphic higher-order functions. In *International ACM Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 191–202, 2004.

[42] Samin Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 14–26, London, United Kingdom, 2001.

[43] Bart Jacobs and Erik Poll. Java program verification at nijmegen: Developments and perspective. In Kokichi Futatsugi, Fumio Mizoguchi, and Naoki Yonezaki, editors, *ISSS*, volume 3233 of *LNCS*, pages 134–153. Springer, 2003.

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

[45] Gerwin Klein, Philip Derrin, and Kevin Elphinstone. Experience report: seL4: formally verifying a high-performance microkernel. In Graham Hutton and Andrew P. Tolmach, editors, *ICFP*, pages 91–96. ACM, 2009.

[46] 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 (22nd SOSP'09), Operating Systems Review (OSR), pages 207–220, Big Sky, MT, 2009. ACM SIGOPS.

[47] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 42–54, 2006.

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

[49] Pierre Letouzey. *Programmation fonctionnelle certifiÃ©e – l'extraction de programmes dans l'assistant Coq.* PhD thesis, UniversitÃ© Paris 11, 2004.

[50] Panagiotis Manolios and J Strother Moore. Partial functions in ACL2. 2003.

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

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

[53] Nicolas Marti, Reynald Affeldt, and Akinori Yonezawa. Verification of the heap manager of an operating system using separation logic. In *Proceedings of the Third SPACE*, pages 61–72, Charleston, SC, USA, 2006.

[54] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[55] Andrew McCreight. Practical tactics for separation logic. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLs*, volume 5674 of *LNCS*, pages 343–358. Springer, 2009.

[56] Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. In Franz Baader, editor, *Automated Deduction - CADE-19, 19th International Conference on Automated Deduction Miami Beach, FL, USA, July 28 - August 2, 2003, Proceedings*, volume 2741 of *LNCS*, pages 121–135. Springer, 2003.

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

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

[59] Magnus O. Myreen. Separation logic adapted for proofs by rewriting. In Matt Kaufmann and Lawrence C. Paulson, editors, *ITP*, volume 6172 of *LNCS*, pages 485–489. Springer, 2010.

[60] Magnus O. Myreen and Michael J. C. Gordon. Verified LISP implementations on ARM, x86 and powerPC. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLs*, volume 5674 of *LNCS*, pages 359–374. Springer, 2009.

[61] A. Nanevski and G. Morrisett. Dependent type theory of stateful higher-order functions. Technical report, Citeseer, 2005.

[62] Aleksandar Nanevski, Greg Morrisett, Avi Shinnar, Paul Govereau, and Lars Birkedal. Ynot : Reasoning with the awkward squad. In *ACM International Conference on Functional Programming (ICFP)*, 2008.

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

[64] Aleksandar Nanevski, Viktor Vafeiadis, and Josh Berdine. Structuring the verification of heap-manipulating programs. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 261–274. ACM, 2010.

[65] Zhaozhong Ni and Zhong Shao. Certified assembly programming with embedded code pointers. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2006.

[66] Peter O'Hearn and David Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

[67] Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Proceedings of CSL'01*, volume 2142 of *LNCS*, pages 1–19, Berlin, 2001. Springer-Verlag.

[68] Peter W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1–3):271–307, 2007.

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

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

[71] Matthew J. Parkinson and Gavin M. Bierman. Separation logic, abstraction and inheritance. In George C. Necula and Philip Wadler, editors, *POPL*, pages 75–86. ACM, 2008.

[72] Matthew J. Parkinson, Richard Bornat, and Peter W. O'Hearn. Modular verification of a non-blocking stack. In Martin Hofmann and Matthias Felleisen, editors, *POPL*, pages 297–302. ACM, 2007.

[73] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, 1994.

[74] Alexandre Pilkiewicz and FranÃ§ois Pottier. The essence of monotonic state. In *Proceedings of the Sixth ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI'11)*, Austin, Texas, 2011.

[75] Fran cois Pottier. Hiding local state in direct style: A higher-order anti-frame rule. In *LICS*, pages 331–340. IEEE Computer Society, 2008.

[76] Yann Régis-Gianas and François Pottier. A Hoare logic for call-by-value functional programs. In *International Conference on Mathematics of Program Construction (MPC)*, 2008.

[77] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 55–74, 2002.

[78] K. Rustan, M. Leino, and M. Moskal. VACID-0: Verification of Ample Correctness of Invariants of Data-structures, Edition 0. 2010.

[79] Peter H. Schmitt, Mattias Ulbrich, and Benjamin Weiß. Dynamic frames in Java dynamic logic. In Bernhard Beckert and Claude Marché, editors, *Proceedings, International Conference on Formal Verification of Object-Oriented Software (FoVeOOS 2010)*, volume 6528 of *LNCS*, pages 138–152. Springer, 2011.

[80] Jan Schwinghammer, Hongseok Yang, Lars Birkedal, FranÃ§ois Pottier, and Bernhard Reus. A semantic foundation for hidden state. In C.-H. L. Ong, editor, *Proceedings of the 13th International Conference on Foundations of Software Science and Computational Structures (FOSSACS 2010)*, volume 6014 of *LNCS*, pages 2–17. Springer, 2010.

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

[82] Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In Martin Hofmann and Matthias Felleisen, editors, *POPL*, pages 97–108. ACM, 2007.

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

[84] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS*. The Internet Society, 2000.

[85] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Scalable shape analysis for systems code. In Aarti Gupta and Sharad Malik, editors, *CAV*, volume 5123 of *LNCS*, pages 385–398. Springer, 2008.

[86] Hongseok Yang and Peter W. O'Hearn. A semantic basis for local reasoning. In Mogens Nielsen and Uffe Engberg, editors, *FoSSaCS*, volume 2303 of *LNCS*, pages 402–416. Springer, 2002.

[87] Nobuko Yoshida, Kohei Honda, and Martin Berger. Logical reasoning for higher-order functions with local state. In *International Conference on Foundations of Software Science and Computation Structures (FOSSACS)*, volume 4423 of *LNCS*, pages 361–377. Springer, 2007.

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