J. Functional Programming **13** (4): 709–745, July 2003. Printed in the United Kingdom

Verification of Non-Functional Programs using Interpretations in Type Theory

Jean-Christophe Filliâtre∗
LRI, Université Paris Sud, France
(*e-mail:* `filliatr@lri.fr`)

Abstract

We study the problem of certifying programs combining imperative and functional features within the general framework of type theory.

Type theory is a powerful specification language, which is naturally suited for the proof of purely functional programs. To deal with imperative programs, we propose a logical interpretation of an annotated program as a partial proof of its specification. The construction of the corresponding partial proof term is based on a static analysis of the effects of the program which excludes aliases. The missing subterms in the partial proof term are seen as proof obligations, whose actual proofs are left to the user. We show that the validity of those proof obligations implies the total correctness of the program.

This work has been implemented in the Coq proof assistant. It appears as a tactic taking an annotated program as argument and generating a set of proof obligations. Several nontrivial algorithms have been certified using this tactic.

∗ This article was written while the author was International Fellow at Computer Science Laboratory, SRI International, Menlo Park, CA.

1 Introduction

This paper is a presentation of the author's thesis (Filliâtre, 1999b). It proposes a new approach to the certification of imperative programs, which combines old ideas about software correctness with recent work on logical frameworks.

A formal method to establish software correctness can involve several steps. The first one is the *specification*. A second one is a method to *generate some proof obligations*. And a third one is a framework to *establish their validity*. The second step, which expresses what must be proved to establish the correctness, has been widely studied. Pioneers were R. W. Floyd (Floyd, 1967), C. A. R. Hoare (Hoare, 1969) and E. W. Dijkstra (Dijkstra, 1976), quickly followed by many others. Some methods derive the proof obligations from the programs, following Floyd-Hoare logic (Jones, 1980; Reif, 1995), and others derive them from specifications, following E. W. Dijkstra (Back, 1981; Morgan, 1990; Abrial, 1996). But almost nothing has been done regarding the other two steps, where standard mathematics were assumed most of time. And although the proof of a program can be conducted on paper mathematically, as has been done brilliantly by C. A. R. Hoare in (Hoare, 1971), formal specification languages and formal proof tools were expected and still had to be defined.

Formal logical frameworks appeared later and, paradoxically, independently of the software correctness concern. They were rather a rehabilitation of logic and λ-calculus, deprecated by traditional mathematicians. The first implementation of a logic was N. de Bruijn's Automath (de Bruijn, 1980), followed by many others, including Nqthm, PVS, HOL, Nuprl and Coq. Some of them implement highly expressive logics, whose counterpart is a relatively poor automation. The system Coq implements the Calculus of Inductive Constructions, an extension of J.-Y. Girard's system F (Girard, 1972) developed by G. Huet, T. Coquand and C. Paulin (Coquand & Huet, 1988; Paulin-Mohring, 1989b), which belongs to the family of type theories.

Type theory identifies types with propositions and terms with proofs, through the widely known Curry-Howard isomorphism. There is no real difference between the usual first-order objects of the mathematical discourse — such as naturals, sets and so forth — and the proof objects. The natural 2 is a first-order object of type nat, and a proof that 2 is even is a first-order object of type $even(2)$. One can define a function f taking as arguments a natural n and a proof that n is even, and its type would be something like $\forall n : \mathsf{nat}.\ even(n) \rightarrow \tau$. Such a function represents a *partial function* on naturals, where the proof of $even(n)$ may be seen as a *precondition*. Similarly, one can define a function returning a proof term. For instance, the function f could return a natural p and a proof that $n = 2 \times p$. Finally, the type of f will look like $\forall n : \mathsf{nat}.\ even(n) \rightarrow \exists p : \mathsf{nat}.\ n = 2 \times p$, where the proof of $n = 2 \times p$ may be seen as a *postcondition*. More generally, a type of the form

$$\forall x : \mathsf{nat}.\ P(x) \rightarrow \exists y : \mathsf{nat}.\ Q(x, y) \tag{1}$$

is the type of a function with a precondition P and a postcondition Q. Building a term of this type is exactly like building a function together with a proof of its correctness, and consequently type theory appears as naturally suited for the proof of purely functional programs. Moreover, there is a systematic way to extract the underlying program from such a proof, as has been demonstrated by C. Paulin (Paulin-Mohring, 1989a; Paulin-Mohring, 1989b). Conversely, C. Parent showed that there is a way to partly reconstruct a proof of (1) from a given functional program of the right type, which leads to the expected proof obligations (Parent, 1993; Parent, 1995).

Our objective is to cope with imperative programs in this context. We propose an interpretation of the Hoare triple $\{P\}\ e\ \{Q\}$ as a proof of the above proposition (1), and then define a systematic construction of this proof from a given annotated program, where the lacking proof terms are the so-called proof obligations. We still have to give the precise meaning of x and y in this interpretation. Obviously, they are the *input* and *output* of the program, but there are many ways to represent them. The tradition in denotational semantics is to see them as memory states, often called *stores*. Although it is suitable from a semantical point of view, it is far from being natural, and is not practical, when we try to elaborate the correctness proof of a program. Indeed, when doing the proof on a sheet of paper, we do not use a store, but mathematical variables to represent the values of the program variables, in the same way as when translating a piece of code from an imperative style to a purely functional one. Thus, we choose to represent the input and output of the program by *tuples of values* representing the values of the variables involved in the computation.

Outline. The main steps of our method are the following. First, Section 2 introduces a programming language with logical annotations, which mixes functional and imperative features. A notion of typing with effects is given for this language, following the work of J.-P. Talpin and P. Jouvelot (Talpin & Jouvelot, 1994). The purpose of effects inference is two fold: it determines the variables implicated in the computation and that should appear therefore in the interpretation; and it excludes programs containing aliases, because such programs do not have any natural functional interpretation. This static analysis is done recursively over the structure of the program, giving a type and an effect to each of its subexpressions.

Then, Section 3 exploits the types and effects informations to build an interpretation in the Calculus of Inductive Constructions of an annotated program e, as a partial proof term $\widehat{e}$ of a proposition expressing the specification of the program in a functional way. The missing parts in the proof term $\widehat{e}$ are the obligations, left to the user. This interpretation is inspired by the call-by-value translations introduced by E. Moggi (Moggi, 1991) and P. Wadler (Wadler, 1993). Some examples are given to illustrate the generation of proof obligations in three important situations — assignment, function call with side effects and while loop.

Section 4 establishes the correctness of the method, namely that if all proof obligations in $\widehat{e}$ are fulfilled by the user, then the program e satisfies its specification. We proceed in three steps: first, we define an operational semantics for our language, borrowed from A. K. Wright and M. Felleisen's proof of type soundness for ML with references (Wright & Felleisen, 1994). Then, we show that the functional program underlying our interpretation, obtained by erasing the logical parts in $\widehat{e}$ following C. Paulin's extraction (Paulin-Mohring, 1989b; Paulin-Mohring, 1989a), is semantically equivalent to the initial program e. Finally, we define the notion of correctness for the annotated programs and we relate this notion to the one of realizability associated to extraction; it leads to the correctness result of our method.

Last, Section 5 gives some details about the implementation of this method in the Coq proof assistant, discusses some completeness issues and describes the case studies developed by the author. Section 6 discusses possible extensions and compares our work with other approaches.

This paper is intended to be a stand-alone presentation, and does not assume any knowledge of the author's thesis (Filliâtre, 1999b). However, the interested reader will find in (Filliâtre, 1999b) detailed proofs of the results given here, some slight generalizations, and a comprehensive presentation of some case studies.

2 A programming language with annotations

In introducing a programming language with annotations, we first define a notion of *annotated type*. Then the constructs of *annotated programs* are introduced and discussed. Finally, a *typing system* for programs is given, which includes an effect inference.

2.1 Types, effects and specifications

Types in programs are a first level of specification. Knowing that a function f has type $\mathsf{int} \rightarrow \mathsf{int}$ prevents us from applying it to a boolean. Types express simple properties of the programs that can be checked by the compiler, and even inferred in some programming languages. An important point is that a good typing system allows *separate compilation*: the type of a function f is the only information needed in compiling a program that uses f.

Obviously, simple types are not enough to guarantee the correct use of a function. If f accepts only even numbers, for instance, then it must be specified and checked for any use of this function. In the general case, this is no longer checkable by the compiler. But we can keep the idea of separate compilation and ask what information is required to use a function. The specification is clearly part of it. In the example above, it would be a precondition expressing that the argument of f must be even. It could also be a postcondition expressing a property of the result of the function call.

Side effects must clearly be part of the specification as well. Indeed, the two computations $f(0)+f(0)$ and $2\times f(0)$ are not equivalent as soon as f has some side effect, such as the modification of a global reference. A possibility is to make explicit side effects as a proposition in the postcondition of f, such as $y = \overleftarrow{y} + 1 \wedge \forall z.z \not\equiv y \Rightarrow z = \overleftarrow{z}$, where $\overleftarrow{x}$ denotes the value of the reference x before the function call. But we expect a more implicit handling of side effects. We would like to declare that f modifies the contents of the reference y, and nothing else, the invariance of the other references being an implicit consequence. We also expect the side effects of a function to be inferred from its definition, or at least checked.

Finally, the notions of pre- and postconditions must be made precise. Our specification language will be the Calculus of Inductive Constructions (CIC for short) (Coquand & Huet, 1988; Paulin-Mohring, 1989b), which is a typed λ-calculus extending the system F with higher order, dependent types and inductive definitions. The terms of CIC and its typing rules are given in the appendix. The typing judgment is written $\Gamma \vdash_{\text{CIC}} u : t$, where Γ is a typing environment and u, t are terms. Terms of type Prop are called *logical propositions* in the following. We assume the base types of our language to be definable in CIC.

In consequence, our notion of *annotated type* is made of a type, an effect and a specification.

First, we need to define the precise notion of *effect*. Extensive work has been done on the static analysis of programs, among which *The Type and Effect Discipline* of J.-P. Talpin and P. Jouvelot (Talpin & Jouvelot, 1994) is probably the most well known. We took some inspiration from their work, but our notion of effect is much simpler, for two reasons. First, we do not consider aliases in programs, and therefore we do not need the notion of regions. Second, we do not consider effect polymorphism — the possibility to abstract the type of a function with respect to an effect — and hence we do not need effect variables. We will justify those restrictions in Section 2.3.

In our case, an effect is a pair of two sets of variables, the first one representing the references possibly accessed by the program, and the second one the references possibly modified by the program.

Definition 1 (*effects*)
An effect is a pair $\epsilon = (\rho, \omega)$ where ρ and ω are two finite sets of variables such that $\omega \subseteq \rho$. We will write $\bot$ for the empty effect, that is, $(\emptyset, \emptyset)$.

The most commonly used operation on effects is their *union*. It appears naturally when two program expressions follow in a sequence. This binary operation, written $\sqcup$, is defined by

$$(\rho_1, \omega_1) \sqcup (\rho_2, \omega_2) \quad \stackrel{\text{def}}{=} \quad (\rho_1 \cup \rho_2, \omega_1 \cup \omega_2)$$

We will also need the operation removing a variable x from an effect ϵ, written $\epsilon \backslash x$, defined by

$$\epsilon \backslash x \quad \stackrel{\text{def}}{=} \quad (\rho \backslash \{x\}, \omega \backslash \{x\})$$

Then we can introduce the notions of *pre-* and *postcondition*. A precondition is a logical proposition which may refer to the current values of the references contained in the environment. The current value of the reference x will be referred to directly as x. A postcondition is a proposition which may also refer to the current values of the references, still with the same notation, but also to the values of the references *before* the evaluation of the program. This value of the reference x will be written $\overleftarrow{x}$.

Finally, we can introduce the notion of *annotated types*. As has been sketched in the introduction of this section, the information characterizing a computation is the type of the returned value, an effect and a specification. A type may be a base type, like int or bool. It may also be the type of a reference containing a value of type τ, which will be written τ ref. The last possibility is the type of a function taking a value of some given type and returning a computation. It leads naturally to the definitions of types for values and types for computations, in a mutually recursive way.

Definition 2 (*annotated types*)
The type expressions for values and computations are mutually recursively defined by the following grammar:

$$\begin{cases} \textit{values} & \tau & ::= & \beta \mid (x:\tau) \rightarrow \kappa \mid \tau\ \mathsf{ref} \\ \textit{computations} & \kappa & ::= & (r:\tau, \epsilon, P, Q) \end{cases}$$

where β is a functional base type, P a precondition and Q a postcondition. In the value type $(x : \tau) \rightarrow \kappa$ the variable x is bound in κ, and in the computation type $(r : \tau, \epsilon, P, Q)$ the variable r is bound in Q. If a type τ does not contain the construct ref, it is said to be pure, which we will write "τ pure".

Environments are lists of bindings of types to variables, as usual. If Γ is an environment, we will write $x : \tau \in \Gamma$ to express that there exist some environments Γ_1 and Γ_2 such that $\Gamma = \Gamma_1, x : \tau, \Gamma_2$, where x is not bound in Γ_2. In the following, we will often need to refer to the sets of references of an environment, which is defined by

$$\textit{Refs}(\Gamma) \quad \stackrel{\text{def}}{=} \quad \{\ x \mid \exists\tau.\ x : \tau\ \mathsf{ref} \in \Gamma\ \}$$

For convenience, we also introduce a notation for the set of all references appearing in a precondition P and a postcondition Q, given an environment Γ:

$$\textit{AllV}(\Gamma, P, Q) \quad \stackrel{\text{def}}{=} \quad \Big(FV(P) \cup FV(Q) \cup \{\, x \mid \overleftarrow{x} \in FV(Q)\,\}\Big) \cap \textit{Refs}(\Gamma)$$

To type check the pre- and postconditions, we have to define the logical environments in which they are going to be typed. It means that we must define what can be mentioned in a pre- or a postcondition. First, all predicates, like equality, and all purely functional operations, like addition, will be available for building the annotations. Second, we need to express properties of (i) the values contained in the references, and (ii) the values returned by the programs. The main difficulty occurs when the annotations need to use the functions present in the environment or contained in references. Indeed, if f is a function which modifies its argument in place, what is the meaning of a proposition like $f(x) = x + 1$ in the specification? Does x on the right side mention the value of x before the function call, or after?

Although it is possible to cope with this problem by considering a logical interpretation of functions in the annotations, see (Filliâtre, 1999b), we will consider here a simpler case where the annotations can only mention values of base types. Notice this eliminates the possibility of mentioning functions *in the annotations* and obviously not the possibility of using them in the programs. Then, if Γ is an environment whose references containing values of base types are $x_1 : \beta_1$ ref, $\dots, x_n : \beta_n$ ref, we define the environments of pre- and postcondition by

$$Pre(\Gamma) \stackrel{\text{def}}{=} x_1 : \beta_1, \dots, x_n : \beta_n$$

and

$$\begin{array}{lll} Post(\Gamma, r : \tau) & \stackrel{\text{def}}{=} & Pre(\Gamma), \overleftarrow{x_1} : \beta_1, \dots, \overleftarrow{x_n} : \beta_n, r : \beta \qquad \text{if } \tau = \beta \text{ or } \tau = \beta \text{ ref} \\ Post(\Gamma, r : \tau) & \stackrel{\text{def}}{=} & Pre(\Gamma), \overleftarrow{x_1} : \beta_1, \dots, \overleftarrow{x_n} : \beta_n \qquad \text{otherwise} \end{array}$$

These environments being defined, we are in a position to type check the pre- and postconditions, and so to check that an annotated type is well formed.

Definition 3 (*well formed annotated types*)
The judgments $\Gamma \vdash_a \tau$ wf and $\Gamma \vdash_a \kappa$ wf are inductively defined over the structure of τ and κ by the following set of rules:

$$\frac{}{\Gamma \vdash_a \beta \ \mathsf{wf}} \qquad \frac{\Gamma \vdash_a \tau_1 \ \mathsf{wf} \qquad \Gamma, x : \tau_1 \vdash_a \kappa_2 \ \mathsf{wf}}{\Gamma \vdash_a (x : \tau_1) \rightarrow \kappa_2 \ \mathsf{wf}} \qquad \frac{\Gamma \vdash_a \tau \ \mathsf{wf} \qquad \tau \ \mathsf{pure}}{\Gamma \vdash_a \tau \ \mathsf{ref} \ \mathsf{wf}}$$

$$\frac{\begin{array}{c}\Gamma \vdash_a \tau \ \mathsf{wf} \qquad \omega \subseteq \rho \subseteq \mathit{Refs}(\Gamma) \qquad \mathit{AllV}(\Gamma, P, Q) \subseteq \rho \\ Pre(\Gamma) \vdash_{\mathrm{CIC}} P : \mathsf{Prop} \qquad Post(\Gamma, r : \tau) \vdash_{\mathrm{CIC}} Q : \mathsf{Prop}\end{array}}{\Gamma \vdash_a (r : \tau, (\rho, \omega), P, Q) \ \mathsf{wf}}$$

Well formedness of an environment Γ is inductively defined by

$$\frac{}{\emptyset \ \mathsf{wf}} \qquad \frac{\Gamma \ \mathsf{wf} \quad \Gamma \vdash_a \tau \ \mathsf{wf}}{\Gamma, x : \tau \ \mathsf{wf}}$$

To summarize, our notion of annotated type includes the usual notion of type, an effect and a specification as a pre- and a postcondition. For instance, assuming that we have also a type τ array for arrays of values of type τ, one predicate $\mathit{sorted} : \tau \ \mathsf{array} \rightarrow \mathsf{Prop}$ expressing that an array is sorted and one predicate $\mathit{permut} : \tau \ \mathsf{array} \rightarrow \tau \ \mathsf{array} \rightarrow \mathsf{Prop}$ expressing that two arrays are permutations of each other, the annotated type of an in place sorting algorithm would be the following:

$$\kappa \quad = \quad (t : \tau \ \mathsf{array}) \rightarrow (\mathsf{unit}, (\{t\}, \{t\}), \mathsf{True}, \mathit{sorted}(t) \wedge \mathit{permut}(t, \overleftarrow{t}))$$

This is the type of a function taking an array t as argument and returning no value (i.e., the value () of type unit), modifying the contents of t (t appears in the input as well as in the output), with no precondition (i.e., the proposition True) and a postcondition expressing that the final value of t is a sorted array, which is a permutation of the initial value of t.

Annotated types are defined; we can introduce now the annotated programs.

2.2 Annotated programs

We consider a programming language with both imperative and functional features. The constructs common to both worlds include the constants, the conditional, the

$$\begin{array}{lcl} e & ::= & \{P\}\ s\ \{r \mid Q\} \\ s & ::= & c \mid x \mid (e\ e) \mid \mathsf{fun}\ (x : \tau) \rightarrow e \mid \mathsf{rec}\ f\ (\vec{x} : \vec{\tau}) : \kappa\ \{\mathsf{variant}\ \nu\} = e \\ & \mid & \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid e\ ;\ e \mid \mathsf{while}\ e\ \mathsf{do}\ \{\mathsf{invariant}\ P\ \mathsf{variant}\ \nu\}\ e\ \mathsf{done} \\ & \mid & \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \mathsf{ref}\ e \mid\ !x \mid x\ :=\ e \end{array}$$

Fig. 1. *Abstract syntax of annotated programs*

function definition and the function call. As an imperative language, it contains references, sequences and loops. As a functional language, it also contains functions as first-order values (passed as arguments and returned by other functions), partial application and a let in construct. This last construct allows binding of a new reference in an expression, and thus provides local variables. This language has a call-by-value semantics, where the argument of a function is evaluated before the function itself and multiple arguments are evaluated from right to left. A formal semantics of the language is given in Section 4.1.

Termination. Since we are interested only in total correctness, we must be in a position to justify the termination of programs, that is, of loops and recursive functions. Such a justification usually involves a quantity, the variant, which strictly decreases for some well founded relation. Instead of limiting the variant to a non negative integer, or an ordinal, as is usually done, we will let the user specify a variant as a pair $\nu = (\phi, R)$ where ϕ is the quantity itself and R the relation, that he or she will have to prove to be well founded.

Definition 4 (*variant*)
Γ being a well formed environment, a *variant* in Γ is a pair $\nu = (\phi, R)$, well typed in $Pre(\Gamma)$, whose type has the form

$$Variant(A) \quad \stackrel{\text{def}}{=} \quad A \times (A \rightarrow A \rightarrow \mathsf{Prop})$$

where A is any type.

Then, loops and recursive functions will be explicitly annotated by such a variant. In the case of a recursive function, the variant is usually defined in terms of the function's arguments — otherwise, we would have written a loop — and the syntax of recursive functions introduces the variant after the arguments.

We are now in position to introduce the syntax of annotated programs.

Definition 5 (*annotated programs*)
The abstract syntax of annotated programs is given in Figure 1, where P is a precondition, Q a postcondition and ν a variant.

One can notice that references are handled only as *variables*, in the constructs $!x$ and $x\ :=\ e$, while the tradition in functional programming languages with references is to allow any expression of type τ ref to be dereferenced or assigned. This will be justified in the next section.

A program, and any of its subexpressions, is annotated with a pre- and postcondition, in the tradition of Floyd-Hoare logic. The syntax is the following:

$$\{P\}\ s\ \{r \mid Q\}$$

where r binds the result of s in Q. In the case of a function $\mathsf{fun}\ (x : \tau) \to e$, however, the annotation will usually not constrain the function itself but the body e directly, with the syntax

$$\mathsf{fun}\ (x : \tau) \to \{P\}\ s\ \{r \mid Q\}$$

Loop invariants. Among program annotations, a particular emphasis is usually made on *loop invariants.* Indeed, due to the structure of the loop, which does not return a value but modifies some data in place, there is usually a property which persists at each iteration of the loop, called the *loop invariant.* We could use the notations already introduced to annotate a loop body e_2 in the following way:

$$\mathsf{while}\ e_1\ \mathsf{do}\ \{\mathsf{variant}\ \nu\}\ \{P\}\ s_2\ \{Q\}\ \mathsf{done}$$

but, since usually P and Q are of the form $I \wedge T_1$ and I, where T_1 is a proposition expressing that the test e_1 is satisfied, we will introduce a particular notation of the kind

$$\mathsf{while}\ e_1\ \mathsf{do}\ \{\mathsf{invariant}\ I\ \mathsf{variant}\ \nu\}\ e_2\ \mathsf{done}$$

Here is an example of such an invariant on a loop computing in the reference s the sum of the integers from 0 up to the value initially contained in the reference n:

$$
\begin{array}{l}
s\ :=\ 0; \\
\mathsf{while}\ !n > 0\ \mathsf{do} \\
\quad \{\ \mathsf{invariant}\ s = \sum_{i=n+1}^{i=\overleftarrow{n}} i\ \ \mathsf{variant}\ (n, <)\ \} \\
\quad s\ :=\ !s\ +\ !n; \\
\quad n\ :=\ !n\ -\ 1 \\
\mathsf{done} \\
\{\ s = \sum_{i=0}^{i=\overleftarrow{n}} i\ \}
\end{array}
$$

Another example is given later (Example 3 on page 727).

Contrary to a loop, a recursive function usually has distinct pre- and postconditions, and its annotation will look like

$$\mathsf{rec}\ f\ (\vec{x} : \vec{\tau}) : \kappa\ \{\mathsf{variant}\ \nu\} = \{P\}\ s\ \{r \mid Q\}$$

similarly to a nonrecursive function. For instance, a recursive function computing the factorial will be annotated as follows:

$$
\begin{array}{l}
\mathsf{rec}\ \mathit{fact}\ (x : \mathsf{nat}) : \kappa\ \{\mathsf{variant}\ (x, <)\} = \\
\quad \{\}\ \mathsf{if}\ x = 0\ \mathsf{then}\ 1\ \mathsf{else}\ x \times (\mathit{fact}\ (x - 1))\ \{r \mid r = x!\}
\end{array}
$$

where $\kappa = (r : \mathsf{nat}, \bot, \mathsf{True}, r = x!)$ is the result type of *fact* (The absence of precondition is understood as the tautological proposition True).

2.3 Typing

In this section, we introduce the typing rules for annotated programs, from which we will get a type checking and effect inference algorithm. Beside the computation of effects, which is not new, the main feature of our typing system is to *exclude programs containing possible aliases*. Indeed, we are looking for a direct interpretation of programs where input variables represent the values of the accessed references and output variables the values of the modified references. Aliases would break this interpretation. Consider, for instance, the following function f which increases its two arguments:

$$f \quad \equiv \quad \mathsf{fun}\ x\ y \rightarrow x := !x + 1;\ y := !y + 1$$

Then its interpretation should be a function taking the values of x and y, let us say v_x and v_y, and returning their new values, namely, $v_x + 1$ and $v_y + 1$. But, if x and y are aliases for the same reference — if f has been applied twice to the same reference, for instance — then the interpretation of f should be now a function taking a single value v and returning $v + 2$. As a consequence, even the type of the interpretation of a function depends on the presence of aliasing.

To avoid any alias introduction, we restrict the program's expressions via typing. An alias is created each time a variable is bound to an existing reference, and such a binding may be realized using the let construct or a function call. We first restrict the use of a variable x designating a reference: it is only allowed in the constructs $!x$, $x := e$ and as the argument of a call-by-reference function. In particular, a reference bound to a variable cannot be bound to another variable using the let construct. In the case of a function call, we avoid aliasing by checking that a function is never applied to a reference which appears already in its effect. Consequently, a global reference modified by a function cannot be passed as an argument to this function and, for the same reason, the same reference cannot be passed twice as arguments of a function.

The typing rules for annotated programs are given below.

Definition 6 (*typing of annotated programs*)
The judgment $\Gamma \vdash_a e : \kappa$ is defined by the following rule:

$$\frac{\Gamma \vdash_a s : (\tau, \epsilon) \qquad \text{let } \epsilon' \stackrel{\text{def}}{=} (AllV(\Gamma, P, Q), \emptyset) \qquad Pre(\Gamma) \vdash_{\text{CIC}} P : \mathsf{Prop} \qquad Post(\Gamma, r : \tau) \vdash_{\text{CIC}} Q : \mathsf{Prop}}{\Gamma \vdash_a \{P\}\ s\ \{r \mid Q\} : (r : \tau, \epsilon \sqcup \epsilon', P, Q)}$$

where the judgment $\Gamma \vdash_a s : (\tau, \epsilon)$ is defined by the inference rules given in Figure 2. In those rules, the judgment $\Gamma \vdash_a e : \kappa$ appears in a weakened form $\Gamma \vdash_a e : (\tau, \epsilon)$ in order to save space, since only types and effects of the subexpressions are needed for the conclusion.

The rules dealing with constants, abstractions, conditionals, sequences and loops are immediate. The three rules dealing directly with references — creation, access and assignment — are also self-explanatory. The key rules, which exclude aliasing, are the rule for variables, the two rules for function calls and the two rules for the

$$\frac{\mathit{Type}(c) = \tau}{\Gamma \vdash_a c : (\tau, \bot)}(\mathsf{CONST}_a) \qquad \frac{x : \tau \in \Gamma \qquad \tau \ \mathsf{pure}}{\Gamma \vdash_a x : (\tau, \bot)}(\mathsf{VAR}_a)$$

$$\frac{\Gamma, x : \tau \vdash_a e : \kappa \qquad \Gamma \vdash_a \tau \ \mathsf{wf}}{\Gamma \vdash_a \mathsf{fun}\ (x : \tau) \rightarrow e : ((x : \tau) \rightarrow \kappa, \bot)}(\mathsf{FUN}_a)$$

$$\frac{\Gamma \vdash_a e_1 : (\tau_2 \rightarrow (\tau_1, \epsilon), \epsilon_1) \qquad \Gamma \vdash_a e_2 : (\tau_2, \epsilon_2) \qquad \tau_2 \ \mathsf{pure}}{\Gamma \vdash_a (e_1\ e_2) : (\tau_1, \epsilon_1 \sqcup \epsilon_2 \sqcup \epsilon)}(\mathsf{APP}_a)$$

$$\frac{\Gamma \vdash_a e : ((x : \tau_1 \ \mathsf{ref}) \rightarrow (\tau, \epsilon), \epsilon_2) \qquad r : \tau_1 \ \mathsf{ref} \in \Gamma \qquad r \notin (\tau, \epsilon)}{\Gamma \vdash_a (e\ r) : (\tau[x \leftarrow r], \epsilon_2 \sqcup \epsilon[x \leftarrow r])}(\mathsf{APPREF}_a)$$

$$\frac{\begin{array}{c}\Gamma \vdash_a \vec{\tau} \ \mathsf{wf} \qquad \Gamma, \vec{x} : \vec{\tau} \vdash_a \kappa \ \mathsf{wf} \\ \Gamma, f : (\vec{x} : \vec{\tau}) \rightarrow \kappa, \vec{x} : \vec{\tau} \vdash_a e : \kappa \qquad \mathit{Pre}(\Gamma, \vec{x} : \vec{\tau}) \vdash_{\mathrm{CIC}} \nu : \mathit{Variant}(A)\end{array}}{\Gamma \vdash_a \mathsf{rec}\ f\ (\vec{x} : \vec{\tau}) : \kappa\ \{\mathsf{variant}\ \nu\} = e : ((\vec{x} : \vec{\tau}) \rightarrow \kappa, \bot)}(\mathsf{REC}_a)$$

$$\frac{\Gamma \vdash_a e_1 : (\mathsf{bool}, \epsilon_1) \qquad \Gamma \vdash_a e_2 : (\tau, \epsilon_2) \qquad \Gamma \vdash_a e_3 : (\tau, \epsilon_3)}{\Gamma \vdash_a \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : (\tau, \epsilon_1 \sqcup \epsilon_2 \sqcup \epsilon_3)}(\mathsf{COND}_a)$$

$$\frac{\Gamma \vdash_a e_1 : (\tau_1, \epsilon_1) \qquad \tau_1 \ \mathsf{pure} \qquad \Gamma, x : \tau_1 \vdash_a e_2 : (\tau, \epsilon)}{\Gamma \vdash_a \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : (\tau, \epsilon_1 \sqcup \epsilon)}(\mathsf{LET}_a)$$

$$\frac{\Gamma \vdash_a e_1 : (\tau_1 \ \mathsf{ref}, \epsilon_1) \qquad \Gamma, x : \tau_1 \ \mathsf{ref} \vdash_a e_2 : (\tau_2, \epsilon_2) \qquad x \notin \tau_2}{\Gamma \vdash_a \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : (\tau_2, \epsilon_1 \sqcup \epsilon_2 \backslash x)}(\mathsf{LETREF}_a)$$

$$\frac{\Gamma \vdash_a e_1 : (\mathsf{unit}, \epsilon_1) \qquad \Gamma \vdash_a e_2 : (\tau, \epsilon_2)}{\Gamma \vdash_a e_1\ ;\ e_2 : (\tau, \epsilon_1 \sqcup \epsilon_2)}(\mathsf{SEQ}_a)$$

$$\frac{\begin{array}{c}\Gamma \vdash_a e_1 : (\mathsf{bool}, \epsilon_1) \qquad \Gamma \vdash_a e_2 : (\mathsf{unit}, \epsilon_2) \\ \mathit{Pre}(\Gamma) \vdash_{\mathrm{CIC}} P : \mathsf{Prop} \qquad \mathit{Pre}(\Gamma) \vdash_{\mathrm{CIC}} \nu : \mathit{Variant}(A)\end{array}}{\Gamma \vdash_a \mathsf{while}\ e_1\ \mathsf{do}\ \{\mathsf{invariant}\ P\ \mathsf{variant}\ \nu\}\ e_2\ \mathsf{done} : (\mathsf{unit}, \epsilon_1 \sqcup \epsilon_2)}(\mathsf{LOOP}_a)$$

$$\frac{\Gamma \vdash_a e : (\tau, \epsilon) \qquad \tau \ \mathsf{pure}}{\Gamma \vdash_a \mathsf{ref}\ e : (\tau \ \mathsf{ref}, \epsilon)}(\mathsf{REF}_a) \qquad \frac{x : \tau \ \mathsf{ref} \in \Gamma}{\Gamma \vdash_a !x : (\tau, (\{x\}, \emptyset))}(\mathsf{DEREF}_a)$$

$$\frac{x : \tau \ \mathsf{ref} \in \Gamma \qquad \Gamma \vdash_a e : (\tau, (\rho, \omega))}{\Gamma \vdash_a x := e : (\mathsf{unit}, (\{x\} \cup \rho, \{x\} \cup \omega))}(\mathsf{AFF}_a)$$

Fig. 2. *Typing of annotated programs*

let construct. The first one, (VAR_a), does not allow use of a variable as soon as it is a reference — but, of course, it is still possible to use it with the dereference operation, with the assignment or in a function call. The two rules for function calls, (APP_a) and (APPREF_a), distinguish between the case of a functional argument and the case of a call-by-reference. In the latter, a side condition expresses that the reference passed to the function must not be already present in the resulting effects of the function call. Similarly, the two rules for the let bindings, (LET_a) and (LETREF_a), distinguish the case of a purely functional expression and the case of a reference binding. In the latter, a side condition expresses that the reference should not appear in the type of the result: it prevents the binding of a local reference in an abstraction, where the name of this reference in the effects would not represent a valid reference anymore. In the rules (REC_a) and (LOOP_a), the premise $\nu : \mathit{Variant}(A)$ only checks that ν is a well formed variant and does not check that the function or the loop is terminating; it will be done together with correctness in the next section.

A typing algorithm, associated to the above typing rules, realizes type checking and effect inference. Its existence can be formalized as follows:

Proposition 1 (*typing algorithm*)
There exists an algorithm which, given a well formed environment Γ and an annotated program e, terminates and either indicates that e is not typeable in Γ, or returns a (unique) type of computation κ such that $\Gamma \vdash_a e : \kappa$.

Proof
The algorithm proceeds by induction over the structure of e. For any construct, there is only one rule that matches e. If one of the premises is not satisfied, then the algorithm fails and e is not typeable in Γ. Otherwise, the type κ is built in a unique way from the rule and the types returned by the recursive calls, which are unique by induction hypothesis. The termination of the algorithm is justified by the fact that the program's size strictly decreases for each recursive call. $\square$

The above typing algorithm needs some comments:

- Effects are inferred in all cases, except for the bodies of recursive functions; indeed, the type κ in rule (REC_a) contains an effect, which is thus explicitly given. In practice, it is possible to adopt a simpler syntax for recursive functions where only the type of the body is given and where the effect is inferred. However, this is not immediate: the effect of a recursive function must be computed as a least fix-point, and the number of steps may be arbitrarily large. See (Filliâtre, 1999b) for details.
- We made the pre- and postconditions mandatory for each subexpression, mostly to simplify the theoretical presentation. In practice, it would be a burden to introduce so many annotations, most of them being redundant. In the next section, we still assume annotations everywhere, for the simplicity of the definitions, but the examples at the end of the section will be given with a minimum number of annotations. Section 5.3 on case studies will also give an idea of the respective proportions of code and annotations in practice.

3 A logical interpretation

In this section, we define the interpretations of types and programs in the Calculus of Inductive Constructions. The main idea is that an annotated type $\kappa = (r : \tau, (\rho, \omega), P, Q)$ will be translated into a proposition of the form

$$\forall x.\ P(x) \rightarrow \exists (y, r).\ Q(x, y, r)$$

and a program of type κ will be translated into a partial proof term of this proposition. Here x and y will be states containing respectively the values of references from ρ and ω; then $P(x)$ (resp. $Q(x, y, r)$) stands for the precondition P now referencing the values in x (resp. the postcondition Q referencing the values in x and y).

Formally, the terms of the CIC (Coquand & Huet, 1988; Paulin-Mohring, 1993) that we need to consider in this paper obey the following grammar:

$$t \ ::= \ \mathsf{Set} \mid \mathsf{Prop} \mid \mathsf{Type}(i) \mid c \mid x \mid \forall x : t.\ t \mid \lambda x : t.\ t \mid (t\ t)$$

The product $\forall x : t_1.\ t_2$ is written $t_1 \to t_2$ whenever x does not occur free in t_2. The typing judgment is written $\Gamma \vdash_{\mathrm{CIC}} u : t$ and the typing rules are given in the appendix, page 745. We do not need to present inductive types; therefore, the existential quantifier is considered here as a primitive construct, although it can be defined as an inductive type (Paulin-Mohring, 1993). Its type, constructor and elimination are introduced by the following constructs:

$$t \ ::= \ \ldots \mid \exists x : t.\ t \mid (t, t) \mid \mathsf{let}\ (x, x) = t\ \mathsf{in}\ t$$

Similarly to the product, $\exists x : t_1.\ t_2$ is written $t_1 \times t_2$ whenever x does not occur free in t_2. The pair (u_1, u_2) builds a proof of $\exists x : t_1.\ t_2$, where u_1 is the witness of type t_1 and u_2 is of type $t_2[x \leftarrow u_1]$, and a pair of type $t_1 \times t_2$ in the non-dependent case. The $\mathsf{let\ in}$ construct is the corresponding destructor. The typing rules for existential quantification are given in the appendix.

As we already assumed, the pure functional base types of our languages (written β in the previous pages) are among the constants c. And so are the predicates used to specify the programs — like equality, order relations, etc. — and all the purely applicative functions used in the programs — like arithmetical operations, test functions, etc. We also assume the definition of a well founded induction principle[1] WF of type

$$\begin{aligned}\mathsf{WF} : \ & \forall A : \mathsf{Set}.\ \forall R : A \to A \to \mathsf{Prop}.\ (\mathit{well\_founded}\ A\ R) \to \\ & \quad \forall P : A \to \mathsf{Set}. \\ & \qquad (\forall x : A.\ (\forall y : A.\ (R\ y\ x) \to (P\ y)) \to (P\ x)) \to \\ & \qquad \forall a : A.\ (P\ a)\end{aligned}$$

Last, for clarity of the presentation, we assume that CIC contains a primitive notion of records[2]. Record types are written $\{x_1 : t_1; \ldots; x_n : t_n\}$ and their elements are written $\{x_1 = t_1; \ldots; x_n = t_n\}$. If v is such a record, then $v.x$ stands for the value of its field x. We assume that it is possible to abstract a term with respect to a field variable, and to apply a term to a field variable, the corresponding reduction still being the β-reduction[3]. In the following, we need an operation to "update" records with respect to another one. This operation, written $\oplus$, is formally defined as follows: if x and y are two records, then $x \oplus y$ is a record containing all the fields

[1] See for instance the Coq standard library (Coq, 2001).

[2] Although this is not the usual case in implementations, we do show (Filliâtre, 1999b) how to cope with this problem using anonymous tuples.

[3] This assumption allows a nice interpretation of functions taking references as argument. Since our implementation uses anonymous tuples instead of records, this abstraction does not appear explicitly anymore. See (Filliâtre, 1999b) for details.

of x and y and where the value of the field a is $y.a$, if a is a field of y, and $x.a$ otherwise.

3.1 Interpretation of types

Let us define first the interpretation of types in CIC. According to the informal ideas given in the introduction, we expect the interpretation of a specification—that is, a type of computation κ—to be a formula of the kind $\forall x.\ P(x) \rightarrow \exists(y,v).\ Q(x,y,v)$ where x and y are records of values, and v is a value. Since values and computations are mutually recursive, their interpretations depend on each other. The precise definition is given below.

Definition 7 (*interpretation of types*)
The interpretations of types of values and computations in CIC, respectively written $\widehat{\tau}$ and $\widehat{\kappa}$, are mutually recursively defined in a typing environment Γ by

$$\begin{array}{llcll}
\textit{values} & \widehat{\beta} & = & \beta & \\
& \widehat{(x:\tau\ \mathsf{ref}) \rightarrow \kappa} & = & \forall x.\widehat{\kappa} & \text{where } x \text{ is a field} \\
& \widehat{(x:\tau) \rightarrow \kappa} & = & \forall x:\widehat{\tau}.\ \widehat{\kappa} & \text{otherwise} \\
& \widehat{\tau\ \mathsf{ref}} & = & \widehat{\tau} & \\
\textit{computations} & \widehat{(r:\tau,(\rho,\omega),P,Q)} & = & \forall x:\widehat{\rho}.\ P(x) \rightarrow \exists(y,r):\widehat{\omega}\times\widehat{\tau}.\ Q(x,y,r) & \\
\textit{effects} & \widehat{\{x_1,\dots,x_n\}} & = & \{x_1:\widehat{\tau_1};\dots;x_n:\widehat{\tau_n}\} & \text{where } x_i:\tau_i\ \mathsf{ref} \in \Gamma
\end{array}$$

where the propositions $P(x)$ and $Q(x,y,r)$ are formally defined as follows:

$$P(x) \stackrel{\text{def}}{=} P[v \leftarrow x.v]$$

for $v \in FV(P) \cap \mathit{Refs}(\Gamma)$, and

$$Q(x,y,r) \stackrel{\text{def}}{=} Q[\overleftarrow{v} \leftarrow x.v][w \leftarrow x.w][z \leftarrow y.z]$$

for $v \in \{x \mid \overleftarrow{x} \in FV(Q) \wedge x \in \mathit{Refs}(\Gamma)\}$, $w \in (FV(Q) \cap \mathit{Refs}(\Gamma))\backslash\omega$ and $z \in (FV(Q) \cap \mathit{Refs}(\Gamma)) \cap \omega$.

Interpretation of typing environments is then defined as follows:

$$\begin{array}{rcll}
\widehat{x:\tau,\Gamma} & = & x:\widehat{\tau},\ \widehat{\Gamma} & \text{when } \tau \neq \tau'\ \mathsf{ref} \\
\widehat{x:\tau,\Gamma} & = & \widehat{\Gamma} & \text{otherwise}
\end{array}$$

In the above definition, $P(x)$ is defined as the predicate P where all references are replaced by their values in the record x, as expected. For the postcondition Q, the interpretation is a bit more subtle. Indeed, the postcondition may mention the current value of some references which are not modified by the program, that is, which do not appear in ω. Therefore, they must be substituted by their values in the input record x instead of the output record y. This is the reason for the three substitutions in the definition of $Q(x,y,r)$. E.g. if $\mathit{Refs}(\Gamma) = \{u;v\}$, $\omega = \{u\}$ and $Q \equiv u > \overleftarrow{u} \wedge u = v$ then $Q(x,y,r) \equiv y.u > x.u \wedge y.u = x.v$.

Finally, notice that references do not appear anymore in the interpretations of environments: indeed, the interpreted programs do not refer to references anymore but are now functions manipulating their values.

3.2 Interpretation of programs

We define now the interpretation of programs. If e is an annotated program of type κ, then its interpretation $\widehat{e}$ will be a partial term in CIC of type $\widehat{\kappa}$. In this term, all the computational part will be given, and the logical part will appear as proof obligations, written $? : P$, where P is a proposition and '?' a term of type P (i.e., a proof of P) to be given by the user. To build the computational part of $\widehat{e}$ means to construct a functional interpretation of the imperative program e. Due to the strong restrictions we put on the language constructs and on the typing rules to avoid aliasing, such an interpretation is not difficult, and we define it directly. It follows the main ideas of *monadic call-by-value translations*, where monadic lets are used to express the sequentiality of computations.

In the following, we make a slight abuse of notation: if a function f takes a record as argument with fields $l_1, \ldots, l_n$, then we write $(f\ x)$ even for a record x containing fields additional to the l_i's, with the implicit convention that they are forgotten. Similarly, when defining a function returning a record, we allow the return of a record with more fields than the expected ones, the additional ones being implicitly forgotten. Notice that such assumptions would not be necessary in the presence of subtyping.

Finally, we will sometimes omit the domain type in a λ-expression, when obvious from the context, and we will write $\mathsf{let}\ (x, v, q) = e_1\ \mathsf{in}\ e_2$ for $\mathsf{let}\ (y, q) = e_1\ \mathsf{in}\ \mathsf{let}\ (x, v) = y\ \mathsf{in}\ e_2$.

Definition 8 (*interpretation of programs*)
Let Γ be a well formed environment and e a program such that $\Gamma \vdash_a e : \kappa$, with $\kappa = (r : \tau, (\rho, \omega), P, Q)$. Then the interpretation of e in CIC, written $\widehat{e}$, is an incomplete term of type $\widehat{\kappa}$ in the context $\widehat{\Gamma}$, recursively defined on the structure of e in the following way:

$e \equiv \{P\}\ s\ \{Q\}$ with $s = c \mid x$ **:** Then

$$\widehat{e} = \lambda x_0.\lambda p : P(x_0).(\{\}, s, ?_1)$$

with $\widehat{\Gamma}, x_0 : \widehat{\rho}, p : P(x_0) \vdash_{\mathrm{CIC}} ?_1 : Q(x_0, \{\}, s)$. In the following, we will omit the contexts of obligations for clarity. (The context of an obligation is made of $\widehat{\Gamma}$ and of all the variables bound up to its placeholder.)

$e \equiv \{P\}\ (e_2\ e_1)\ \{Q\}$ **:** We know that $\Gamma \vdash_a e_2 : ((x : \tau_1) \rightarrow (\tau, \epsilon, P', Q'), \epsilon_2, P_2, Q_2)$. We distinguish two cases according to the type of e_1:

- If $\Gamma \vdash_a e_1 : (\tau_1, \epsilon_1, P_1, Q_1)$, with $\tau_1 \neq {\_}\ \mathsf{ref}$, then

$$\begin{aligned}\widehat{e} = \lambda x_0.\lambda p : P(x_0).\ & \mathsf{let}\ (x_1, a, q_1) = (\widehat{e_1}\ x_0\ ?_1)\ \mathsf{in} \\ & \mathsf{let}\ (x_2, f, q_2) = (\widehat{e_2}\ (x_0 \oplus x_1)\ ?_2)\ \mathsf{in} \\ & \mathsf{let}\ (x_3, v, q) = (f\ a\ (x_0 \oplus x_1 \oplus x_2)\ ?_3)\ \mathsf{in} \\ & (x_1 \oplus x_2 \oplus x_3, v, ?_4)\end{aligned}$$

with $?_1 : P_1(x_0)$, $?_2 : P_2(x_0 \oplus x_1)$, $?_3 : P'(x_0 \oplus x_1 \oplus x_2)$ and $?_4 : Q(x_0, x_1 \oplus x_2 \oplus x_3, v)$.

- If $e_1 = r : \tau_1'$ ref then

$$\widehat{e} = \lambda x_0.\lambda p : P(x_0).\ \begin{array}{l}\mathsf{let}\ (x_1, f, q_2) = (\widehat{e_2}\ x_0\ ?_1)\ \mathsf{in} \\ \mathsf{let}\ (x_2, v, q) = (f\ r\ (x_0 \oplus x_1)\ ?_2)\ \mathsf{in} \\ (x_1 \oplus x_2, v, ?_3)\end{array}$$

with $?_1 : P_2(x_0)$, $?_2 : P'(x_0 \oplus x_1)$ and $?_3 : Q(x_0, x_1 \oplus x_2, v)$.

$e \equiv \{P\}\ \mathsf{fun}\ (x : \tau_1) \rightarrow e_1\ \{Q\}$ **:** We distinguish two cases:

- If $\tau = \tau_1'$ ref then

$$\widehat{e} = \lambda x_0.\lambda p : P(x_0).(\{\}, \lambda x.\widehat{e_1}, ?_1)$$

with $?_1 : Q(x_0, \{\}, \lambda x.\widehat{e_1})$.

- else

$$\widehat{e} = \lambda x_0.\lambda p : P(x_0).(\{\}, \lambda x : \widehat{\tau}.\widehat{e_1}, ?_1)$$

with $?_1 : Q(x_0, \{\}, \lambda x : \widehat{\tau}.\widehat{e_1})$.

$e \equiv \{P\}\ \mathsf{rec}\ f\ (\vec{x} : \vec{\tau}) : \kappa\ \{\mathsf{variant}\ \nu\} = e_1\ \{Q\}$ **:** We have $\nu = (\phi, R) : \mathit{Variant}(A)$ and $\Gamma, f : (\vec{x} : \vec{\tau}) \rightarrow \kappa, \vec{x} : \vec{\tau} \vdash_a e_1 : (\tau_1, \epsilon_1, P_1, Q_1)$. We build the interpretation of e using the well founded induction principle WF. Since the variant may depend on the current state, we apply WF on the generalized proposition

$$K(\varphi) \quad \stackrel{\text{def}}{=} \quad \forall \vec{x}.\forall x_0.\ \varphi = \phi(\vec{x}, x_0) \rightarrow P_1(x_0) \rightarrow \exists (x_1, r).\ Q_1(x_0, x_1, r)$$

The recursive function is interpreted by the term

$$\begin{array}{rcl}\widehat{f}\ :\ \widehat{(\vec{x} : \vec{\tau}) \rightarrow \kappa} & = & \lambda \vec{x}.\lambda x_0. \\ & & \quad (\mathsf{WF}\ A\ R\ ?_1\ \lambda\varphi.K(\varphi) \\ & & \quad\ \ \lambda\varphi.\lambda f.\lambda\vec{x}.\lambda x_0.\lambda h : \varphi = \phi(\vec{x}, x_0).\ (\widehat{e_1}\ x_0) \\ & & \quad\ \ \phi(\vec{x}, x_0)\ \vec{x}\ x_0\ ?)\end{array}$$

with $?_1 : (\mathit{well\_founded}\ A\ R)$. The unnumbered obligation corresponds to a proof of $\phi(\vec{x}, x_0) = \phi(\vec{x}, x_0)$, which can be automatically inserted. In the interpretation $\widehat{e_1}$, each occurrence of f applied to the arguments $\vec{a}$ and to the state x_i is interpreted by

$$(f\ \phi(\vec{a}, x_i)\ ?_2\ \vec{a}\ x_i\ ?\ ?_3)$$

with $?_2 : (R\ \phi(\vec{a}, x_i)\ \varphi)$ and $?_3 : P_1[\vec{x} \leftarrow \vec{a}](x_i)$. The unnumbered obligation corresponds to a proof of $\phi(\vec{a}, x_i) = \phi(\vec{a}, x_i)$, which can be automatically inserted. Then the interpretation of e itself is defined by

$$\widehat{e} = \lambda x_0.\lambda p : P(x_0).(\{\}, \widehat{f}, ?_4)$$

with $?_4 : Q(x_0, \{\}, \widehat{f})$.
Notice that, for the last two constructs fun and rec, we usually have neither the precondition P nor the postcondition Q, which simplifies the interpretation. Indeed, in practice we are not interested in the function as a value, but we rather specify its *body*, as a computation, the annotations being then the ones of e_1.

$e \equiv \{P\}$ if e_1 then e_2 else e_3 $\{Q\}$ **:**
We have $\Gamma \vdash_a e_1 : (\mathsf{bool}, \epsilon_1, P_1, Q_1)$, $\Gamma \vdash_a e_2 : (\tau, \epsilon_2, P_2, Q_2)$ and $\Gamma \vdash_a e_3 : (\tau, \epsilon_3, P_3, Q_3)$. Then

$$\begin{array}{ll} \widehat{e} = \lambda x_0.\lambda p : P(x_0). & \mathsf{let}\ (x_1, b, q_1) = (\widehat{e_1}\ x_0\ ?_1)\ \mathsf{in} \\ & \mathsf{if}\ b\ \mathsf{then} \\ & \quad \mathsf{let}\ (x_2, v, q_2) = (\widehat{e_2}\ (x_0 \oplus x_1)\ ?_2)\ \mathsf{in}\ (x_0 \oplus x_1 \oplus x_2, v, ?_3) \\ & \mathsf{else} \\ & \quad \mathsf{let}\ (x_2, v, q_3) = (\widehat{e_3}\ (x_0 \oplus x_1)\ ?_4)\ \mathsf{in}\ (x_0 \oplus x_1 \oplus x_2, v, ?_3) \end{array}$$

with $?_1 : P_1(x_0)$, $?_2 : P_2(x_0 \oplus x_1)$, $?_3 : Q(x_0, x_1 \oplus x_2, v)$ and $?_4 : P_3(x_0 \oplus x_1)$.

$e \equiv \{P\}$ let $x = e_1$ in e_2 $\{Q\}$ **:** We have $\Gamma \vdash_a e_1 : (\tau_1, \epsilon_1, P_1, Q_1)$ and $\Gamma, x : \tau_1 \vdash_a e_2 : (\tau, \epsilon_2, P_2, Q_2)$. We distinguish two cases according to the type of e_1:

- If $\tau_1 \neq \_\ \mathsf{ref}$ then

$$\begin{array}{ll} \widehat{e} = \lambda x_0.\lambda p : P(x_0). & \mathsf{let}\ (x_1, x, q_1) = (\widehat{e_1}\ x_0\ ?_1)\ \mathsf{in} \\ & \mathsf{let}\ (x_2, v, q_2) = (\widehat{e_2}\ (x_0 \oplus x_1)\ ?_2)\ \mathsf{in} \\ & (x_1 \oplus x_2, v, ?_3) \end{array}$$

  with $?_1 : P_1(x_0)$, $?_2 : P_2(x_0 \oplus x_1)$ and $?_3 : Q(x_0, x_1 \oplus x_2, v)$.
- If $\tau_1 = \_\ \mathsf{ref}$ then

$$\begin{array}{ll} \widehat{e} = \lambda x_0.\lambda p : P(x_0). & \mathsf{let}\ (x_1, v_0, q_1) = (\widehat{e_1}\ x_0\ ?_1)\ \mathsf{in} \\ & \mathsf{let}\ (x_2, v, q_2) = (\widehat{e_2}\ (x_0 \oplus x_1 \oplus \{x = v_0\})\ ?_2)\ \mathsf{in} \\ & (x_1 \oplus x_2 \backslash x, v, ?_3) \end{array}$$

  with $?_1 : P_1(x_0)$, $?_2 : P_2(x_0 \oplus x_1 \oplus \{x = v_0\})$ and $?_3 : Q(x_0, x_1 \oplus x_2 \backslash x, v)$.

$e \equiv \{P\}$ e_1 ; e_2 $\{Q\}$ **:** The interpretation is similar to the one of a construct let, using the equivalence $e \approx \mathsf{let}\ \_ = e_1\ \mathsf{in}\ e_2$.

$e \equiv \{P\}$ while e_1 do $\{\mathsf{invariant}\ I\ \mathsf{variant}\ \nu\}$ e_2 done $\{Q\}$ **:**
We have $\Gamma \vdash_a e_1 : (r_1 : \mathsf{bool}, \epsilon_1, P_1, Q_1)$ and $\Gamma \vdash_a e_2 : (\mathsf{unit}, \epsilon_2, P_2, Q_2)$. We also have $\nu = (\phi, R) : \mathit{Variant}(A)$. We define $\widehat{e}$ in a way similar to the case of a recursive function, applying WF to the following proposition

$$K(\varphi) \quad \stackrel{\text{def}}{=} \quad \forall x_0.\ \varphi = \phi(x_0) \rightarrow I(x_0) \rightarrow \exists (x_1, r).\ Q(x_0, x_1, r)$$

It leads to the following definition

$$\begin{array}{rcl} \widehat{e} & = & \lambda x_0.\lambda p : P(x_0). \\ & & \quad (\mathsf{WF}\ A\ R\ ?_1\ \lambda\varphi.K(\varphi) \\ & & \quad\ \ \lambda\varphi.\lambda w.\lambda x_0.\lambda h_0 : \varphi = \phi(x_0).\lambda h_1 : I(x_0). \\ & & \quad\quad \mathsf{let}\ (x_1, b, q_1) = (\widehat{e_1}\ x_0\ ?_2)\ \mathsf{in} \\ & & \quad\quad \mathsf{if}\ b\ \mathsf{then} \\ & & \quad\quad\quad \mathsf{let}\ (x_2, \_, q_2) = (\widehat{e_2}\ (x_0 \oplus x_1)\ ?_3)\ \mathsf{in} \\ & & \quad\quad\quad (w\ \phi(x_0 \oplus x_1 \oplus x_2)\ ?_4\ (x_0 \oplus x_1 \oplus x_2)\ ?\ ?_5) \\ & & \quad\quad \mathsf{else} \\ & & \quad\quad\quad (x_0 \oplus x_1, (), ?_6) \\ & & \quad \phi(x_0)\ x_0\ ?\ ?_7) \end{array}$$

with $?_1 : (\mathit{well\_founded}\ A\ R)$, $?_2 : P_1(x_0)$, $?_3 : P_2(x_0 \oplus x_1)$, $?_4 : (R\ \phi(x_0 \oplus x_1 \oplus$

$x_2)\ \varphi)$, $?_5 : I(x_0 \oplus x_1 \oplus x_2)$, $?_6 : Q(x_0, x_0 \oplus x_1, ())$ and $?_7 : I(x_0)$. The unnumbered obligations corresponds to proofs of $\phi(s) = \phi(s)$ for some state s, which can be automatically inserted.

$e \equiv \{P\}$ ref e_1 $\{Q\}$ **:** We have $\Gamma \vdash_a e_1 : (\tau_1, \epsilon_1, P_1, Q_1)$. Then

$$\widehat{e} = \lambda x_0.\lambda p : P(x_0).\mathsf{let}\ (x_1, v, q_1) = (\widehat{e_1}\ x_0\ ?_1)\ \mathsf{in}\ (x_1, v, ?_2)$$

with $?_1 : P_1(x_0)$ and $?_2 : Q(x_0, x_1, v)$.

$e \equiv \{P\}\ !x\ \{Q\}$ **:**

$$\widehat{e} = \lambda x_0.\lambda p : P(x_0).(\{\}, x_0.x, ?_1)$$

with $?_1 : Q(x_0, \{\}, x_0.x)$.

$e \equiv \{P\}\ x\ :=\ e_1\ \{Q\}$ **:** We have $\Gamma \vdash_a e_1 : (\tau_1, \epsilon_1, P_1, Q_1)$. Then

$$\widehat{e} = \lambda x_0.\lambda p : P(x_0).\mathsf{let}\ (x_1, v, q_1) = (\widehat{e_1}\ x_0\ ?_1)\ \mathsf{in}\ (x_1 \oplus \{x = v\}, (), ?_2)$$

with $?_1 : P_1(x_0)$ and $?_2 : Q(x_0, x_1 \oplus \{x = v\}, ())$. $\square$

Some examples should help the reader in understanding how the proof obligations look like.

Example 1. Let us first consider the very simple assignment

$$e \quad \equiv \quad \{x \geq 0\}\ x\ :=\ !x + 1\ \{x > \overleftarrow{x} \geq 0\}$$

in a context Γ containing at least a reference x of type int ref. We have

$$\Gamma \vdash_a e : (\mathsf{unit}, (\{x\}, \{x\}), x \geq 0, x > \overleftarrow{x} \geq 0) \tag{2}$$

and therefore the interpretation of e has type

$$\widehat{e} \quad : \quad \forall x_0 : \{x : \mathsf{int}\}.\ x_0.x \geq 0 \rightarrow \exists (x_1, r) : \{x : \mathsf{int}\} \times \mathsf{unit}.\ x_1.x > x_0.x \geq 0 \tag{3}$$

Following Definition 8, the interpretation of e is the proof term

$$\widehat{e} \quad = \quad \lambda x_0.\ \lambda p : x_0.x \geq 0.\ \mathsf{let}\ v = x_0.x + 1\ \mathsf{in}\ (x_0 \oplus \{x = v\}, (), ? : v > x_0.x \geq 0)$$

After the reduction of this last let in redex, the only proof obligation is the expected one:

$$\forall x.\ x \geq 0 \rightarrow x + 1 > x \geq 0$$

where x stands here for a new variable generalizing $x_0.x$. Thanks to reductions in the proof term $\widehat{e}$, we have found exactly the same proof obligation as the one obtained in Floyd-Hoare logic by combining the rules of consequence and assignment.

Example 2. In the same environment Γ, let us consider now a more complex situation where x is assigned the result of a function call with possible side effects:

$$e \quad \equiv \quad \{x \geq 1\}\ x\ :=\ (f\ 1)\ \{x < \overleftarrow{x}\}$$

Let us assume that f has the following annotated type:

$$f \quad : \quad (y : A) \rightarrow (r : \mathsf{int}, (\{x\}, \{x\}), x \geq y, x = \overleftarrow{x} - y \wedge r = x)$$

We assume that the function call $(f\ 1)$ is annotated in the following way:

$$e \quad \equiv \quad \{x \geq 1\}\ x := \{x \geq 1\}\ (f\ 1)\ \{r \mid x = \overleftarrow{x} - 1 \wedge r = x\}\ \{x < \overleftarrow{x}\}$$

which can be done automatically. (Actually, it is done automatically in the implementation as soon as the function argument is purely functional.) Then the interpretation of e is the proof term

$$\begin{aligned} \widehat{e} \quad = \quad & \lambda x_0.\lambda p : x_0.x \geq 1.\ \mathsf{let}\ (x_1, v, q_f) = \\ & \qquad\qquad \mathsf{let}\ (x_1, r, q_f) = (f\ 1\ x_0\ ?_1)\ \mathsf{in}\ (x_1, r, ?_2) \\ & \qquad\quad \mathsf{in}\ (x_1 \oplus \{x = v\}, \mathsf{void}, ?_3) \end{aligned}$$

where $?_1 : x_0.x \geq 1$, $?_2 : x_1.x = x_0.x - 1 \wedge r = x_1.x$ and $?_3 : v < x_0.x$. The obligation '$?_1$' is trivially discharged by p, since the preconditions of the program and the function f are the same. Similarly the second obligation '$?_2$' is directly established by q_f (its proof *is* q_f), since the function call was precisely annotated with the postcondition of f. Then only obligation '$?_3$' remains. It states that the final postcondition has to hold after the assignment; generalizing $x_0.x$ and $x_1.x$, it can be written as

$$\forall x.\ x \geq 1 \rightarrow \forall x', v.\ x' = x - 1 \wedge v = x' \rightarrow v < x$$

Contrary to Example 1, the value assigned to x is no more substituted but *abstracted* as a variable v; and so is the effect of the function call, abstracted in the variable x'. Then the function postcondition (that is $x' = x - 1 \wedge v = x'$) can be used to establish the final postcondition (that is $v < x$).

Example 3. As a last example, let us consider a very simple while loop which stores in the reference x the least power of 2 greater or equal than a given integer k:

$$\begin{aligned} e \quad \equiv \quad & x\ :=\ 1; \\ & \mathsf{while}\ !x < k\ \mathsf{do} \\ & \quad \{\ \mathsf{invariant}\ \exists i \geq 0.\ x = 2^i \quad \mathsf{variant}\ (2k - x, <_n)\ \} \\ & \quad x\ :=\ 2\ \times\ !x \\ & \mathsf{done} \\ & \{\ x \geq k\ \wedge\ \exists i \geq 0.\ x = 2^i\ \} \end{aligned}$$

where $<_n$ is defined by $x <_n y \equiv 0 \leq x < y$. Let $I(x) \equiv \exists i \geq 0.\ x = 2^i$ be the invariant and $Q(x) \equiv x \geq k\ \wedge\ \exists i \geq 0.\ x = 2^i$ be the postcondition. We have $\Gamma \vdash_a e : (\mathsf{unit}, (\{x\}, \{x\}), \_, Q(x))$ and therefore

$$\widehat{e} \quad : \quad \forall x_0 : \{x : \mathsf{int}\}.\ \exists (x_1, r) : \{x : \mathsf{int}\} \times \mathsf{unit}.\ Q(x_1.x)$$

Regarding annotations, we assume that the postcondition Q of e is also the postcondition of the while loop and that the test $!x < k$ is given the postcondition

$$!x < k \quad \{\ b \mid \mathsf{if}\ b\ \mathsf{then}\ x < k\ \mathsf{else}\ x \geq k\ \}$$

This can be done automatically, at least when the test is functional, and is actually done in the implementation. Then, following Definition 8, we have

$$\begin{array}{rcl} \widehat{e} & = & \lambda x_0.\ \mathsf{let}\ (x_1, v_1) = (x_0 \oplus \{x = 1\}, ())\ \mathsf{in} \\ & & \quad\ \ \mathsf{let}\ (x_2, v_2, q) = (\widehat{w}\ (x_0 \oplus x_1))\ \mathsf{in} \\ & & \quad\ \ (x_1 \oplus x_2, v_2, ?) \end{array}$$

where $\widehat{w}$ is the interpretation of the loop. The only obligation above is immediately discharged by q since the postcondition of the loop is also the postcondition of e. The interpretation of the loop itself is

$$\begin{array}{rcl} \widehat{w} & = & (\mathsf{WF\ int}\ <_n\ ?_1\ \lambda\varphi.K(\varphi) \\ & & \quad \lambda\varphi.\lambda w.\lambda x_0.\lambda h_0 : \varphi = (2k - x_0.x).\lambda h_1 : I(x_0.x). \\ & & \quad\quad \mathsf{let}\ (\{\}, b, q_1) = (\widehat{!x < k}\ x_0)\ \mathsf{in} \\ & & \quad\quad \mathsf{if}\ b\ \mathsf{then} \\ & & \quad\quad\quad \mathsf{let}\ (x_2, \_) = (x_0 \oplus \{x = 2 \times x_0.x\}, ())\ \mathsf{in} \\ & & \quad\quad\quad (w\ (2k - x_2.x)\ ?_2\ (x_0 \oplus x_2)\ ?\ ?_3) \\ & & \quad\quad \mathsf{else} \\ & & \quad\quad\quad (x_0, (), ?_4) \\ & & \quad \phi(x_1)\ x_1\ ?\ ?_5) \end{array}$$

There are five non-trivial proof obligations, which are the following:

- The order relation is well founded: $?_1 : (\mathit{well\_founded}\ \mathsf{int}\ <_n)$
- The variant decreases: $?_2 : \varphi <_n (2k - x_2.x)$.
- The invariant is preserved: $?_3 : I((x_0 \oplus x_2).x)$.
- The postcondition is established when the loop terminates: $?_4 : Q(x_0.x)$
- The invariant is initially true: $?_5 : I(x_1.x)$.

After rewriting φ using h_0, reducing some let in redexes and generalizing over the relevant hypotheses, the four proof obligations above look like

$$\begin{array}{lcl} ?_2 & : & \forall x.\ (\exists i \geq 0.\ x = 2^i) \to x < k \to 0 \leq 2k - 2x < 2k - x \\ ?_3 & : & \forall x.\ (\exists i \geq 0.\ x = 2^i) \to x < k \to \exists i \geq 0.\ 2x = 2^i \\ ?_4 & : & \forall x.\ (\exists i \geq 0.\ x = 2^i) \to x \geq k \to x \geq k\ \wedge\ \exists i \geq 0.\ x = 2^i \\ ?_5 & : & \exists i \geq 0.\ 1 = 2^i \end{array}$$

They are the expected obligations and are all easy to establish.

A practical concern illustrated by these examples may be the size of the proof obligations. As one can deduce from the interpretation, the context of an obligation is proportional to the number of annotations and constructs preceding the obligation point. Thus obligations are never too big for programs of reasonable size. In Section 5.3, we give more detailed figures about several case studies.

Monads and effects. In the above interpretation, there is a genericity which is related to the monadic call-by-value translation. Monads were indeed introduced in computer science by E. Moggi (Moggi, 1991) and P. Wadler (Wadler, 1993) to express the semantics of programming languages within purely functional frameworks.

In (Filliâtre, 1999b), we proposed a generalization of the notion of monad, parameterized by an abstract notion of effect. This allows generic interpretations of several constructs, independently of the nature of the effect. If, for instance, exceptions were added to our programming language, the interpretation of many constructs would stay unchanged if they were defined using the monadic operator. Indeed, the notion of effect would be extended with sets of possibly raised exceptions — following J. Guzmán and A. Suárez (Guzmán & Suárez, 1994) or X. Leroy and F. Pessaux (Leroy & Pessaux, 2000), for instance — and only the monadic operators would have to be redefined. The formal definition of monads parameterized by effects and their properties are given in (Filliâtre, 1999b; Filliâtre, 1999a) and are beyond the scope of this paper.

4 Correctness

We establish the correctness of our method, namely, that if all proof obligations appearing in the interpretation $\widehat{e}$ are completed, then the program e satisfies its specification. We begin by defining a formal semantics for our programs. Then we define the computational contents of the interpretation $\widehat{e}$ and we show that it preserves the semantics of e. Finally, we introduce the notion of *imperative realizability*, which expresses the correctness of the imperative programs. By relating this notion to the usual notion of functional realizability, we establish the correctness of our method.

4.1 Formal semantics

Our interpretation in CIC is actually already a denotational semantics of imperative programs. However, an operational semantics will make the definition of correctness more intuitive and will allow us to state the absence of aliasing in an explicit way. We chose to adopt the operational semantics used by A. K. Wright and M. Felleisen to establish the type soundness of ML with references (Wright & Felleisen, 1994), because it is an intuitive semantics, close to a handmade evaluation of programs. This is a small-step semantics driven by the syntax. Indeed, it consists in a *syntactic* notion of reduction over programs, which use a syntactic distinction between values and expressions, which is the following:

$$\begin{array}{llll} \textit{expressions} & e & ::= & v \mid (e\ e) \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \theta.e \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \\ \textit{values} & v & ::= & c \mid x \mid \mathsf{Y} \mid \lambda x.e \mid \mathsf{ref} \mid\ ! \mid\ := \mid (:=\ v) \\ \textit{states} & \theta & ::= & \{(x,v),\ldots,(x,v)\} \end{array} \tag{4}$$

where λ stands for the usual λ-abstraction and Y for a fixed-point operator. Parentheses are put around applications and only there. The constructs ref, ! and := are not considered as primitive, but directly as functional values, then reducing the number of cases to study in many situations.

A construct $\theta.e$ has been introduced. It represents the program expression e in the *state* θ, a state being a finite mapping from variables—representing references—to

$$
\begin{array}{rcll r}
(c\ v) & \longrightarrow & \delta(c,v) & \text{if } \delta(c,v) \text{ is defined} & (\delta) \\
(\lambda x.e\ v) & \longrightarrow & e[x \leftarrow v] & & (\beta) \\
\mathsf{let}\ x = v\ \mathsf{in}\ e & \longrightarrow & e[x \leftarrow v] & & (let) \\
(\mathsf{Y}\ v) & \longrightarrow & (v\ \lambda x.((\mathsf{Y}\ v)\ x)) & & (Y) \\
\mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 & \longrightarrow & e_1 & & (\textit{if-true}) \\
\mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2 & \longrightarrow & e_2 & & (\textit{if-false}) \\
\\
(\mathsf{ref}\ v) & \longrightarrow & \{(x,v)\}.x & x \text{ fresh}^4 & (ref) \\
\theta.R[!x] & \longrightarrow & \theta.R[v] & (x,v) \in \theta & (deref) \\
\theta.R[x := v] & \longrightarrow & \theta \uplus \{(x,v)\}.R[()] & x \in dom(\theta) & (assign) \\
\theta_1.\theta_2.e & \longrightarrow & \theta_1 \uplus \theta_2.e & & (\theta_{merge}) \\
R[\theta.e] & \longrightarrow & \theta.R[e] \quad \mathsf{if}\ R \neq [] & & (\theta_{lift})
\end{array}
$$

Fig. 3. *Notion of reduction*

values. This mapping is represented as a finite set of pairs (x, v) where x is a variable and v a value. We write $dom(\theta)$ the set of variables mapped in θ. We assume that each variable of $dom(\theta)$ appears exactly once as a first component of an element of θ. Then we write $\theta(x)$ when $x \in dom(\theta)$ to designate the unique element v such that $(x, v) \in \theta$. Finally, we define the operation $\uplus$ on states in the following way:

$$\theta_1 \uplus \theta_2 \quad \stackrel{\text{def}}{=} \quad \{(x,v) \mid (x,v) \in \theta_2 \lor ((x,v) \in \theta_1 \land x \notin dom(\theta_2))\}$$

Before introducing the operational semantics, we can translate our program expressions into the above syntax. First, we get rid of types, which are not involved in the definition of the semantics. Second, some program expressions, such as sequences or loops, can have simpler constructs. So we assume having applied to our programs the following set of translation rules, recursively:

$$
\begin{array}{rcl}
\mathsf{fun}\ x : \tau \rightarrow e & \leadsto & \lambda x.e \\
\mathsf{rec}\ f\ (\vec{x} : \vec{\tau}) : \kappa\ =\ e & \leadsto & (\mathsf{Y}\ \lambda f.\lambda \vec{x}.e) \\
e_1\ ;\ e_2 & \leadsto & \mathsf{let}\ \_ = e_1\ \mathsf{in}\ e_2 \\
\mathsf{while}\ e_1\ \mathsf{do}\ e_2\ \mathsf{done} & \leadsto & ((\mathsf{Y}\ \lambda w.\lambda u.\mathsf{if}\ e_1\ \mathsf{then\ let}\ \_ = e_2\ \mathsf{in}\ (w\ ()) \\
& & \qquad\qquad\qquad\quad \mathsf{else}\ ())\ ())
\end{array}
$$

Then Wright and Felleisen define the *notion of reduction*. There are six reductions for the functional fragment and five for the imperative fragment. The latter need a notion of context R in which the state can be accessed or modified. These contexts are defined by the following grammar:

$$R \quad ::= \quad [] \mid (R\ v) \mid (e\ R) \mid \mathsf{let}\ x = R\ \mathsf{in}\ e \mid \mathsf{if}\ R\ \mathsf{then}\ e\ \mathsf{else}\ e$$

The notion of reduction is given in Figure 3. With respect to (Wright & Felleisen, 1994), we added two rules for the conditional and we modified the rule for assignment since it does not return any value in our case. The reduction of constants assumes an interpretation δ of the constants, as a partial function taking a constant and a value as arguments and returning a value.

[4] x has to be different from all the variables already appearing in this evaluation.

Then, Wright and Felleisen introduce the *evaluation contexts*, defined by

$$E \quad ::= \quad [] \mid (E\ v) \mid (e\ E) \mid \mathsf{let}\ x = E\ \mathsf{in}\ e \mid \mathsf{if}\ E\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \theta.E$$

These contexts set the evaluation order for the different constructs of the language. Finally, the evaluation is the transitive closure of the small-step reduction $\longmapsto$ defined by

$$E[e] \longmapsto E[e'] \quad \text{if and only if} \quad e \longrightarrow e'$$

A key result in (Wright & Felleisen, 1994) is type preservation and we assume this property in the following. The reader should refer to (Wright & Felleisen, 1994) for a type system and a proof of type preservation. Notice although that we are here in a simpler situation since we do not have the delicate issue of polymorphic references.

4.2 Computational contents

The main reason for the correctness of the method is that the computational part of $\widehat{e}$, which computes the output from the input, respects the semantics of e. To express and to establish this property, we give a formal definition to the "computational contents" of a proof term. We follow the work of C. Paulin on extraction and realizability in the Calculus of Inductive Constructions (Paulin-Mohring, 1989a; Paulin-Mohring, 1989b). The set of propositions can be split between *informative* propositions, which are types t such that $\Gamma \vdash_{\text{CIC}} t : \mathsf{Set}$, and *logical* propositions, which are types t such that $\Gamma \vdash_{\text{CIC}} t : \mathsf{Prop}$. Similarly, terms are split between informative and logical ones, depending on whether their types are informative or logical. Then the computational content of informative types and informative terms is introduced by mean of an *extraction* operator, written $\mathcal{E}$, whose definition is given in Figure 4 (see (Paulin-Mohring, 1989b) page 77 for the general definition). This is nothing else than erasure of logical parts.

Then we can formally define the computational contents of the interpretation $\widehat{e}$ as the extraction of this proof term.

Definition 9 (*computational contents*)
Let Γ be a well formed environment and e an annotated program of type κ in Γ. Then we write $\overline{e}$ for the extraction of $\widehat{e}$, that is, a term of type $\mathcal{E}(\widehat{\kappa})$:

$$\overline{e} \quad \stackrel{\text{def}}{=} \quad \mathcal{E}(\widehat{e})$$

The following two theorems express the correctness of the functional interpretation $\overline{e}$. The first one mainly says that if a program e evaluates to v then the functional interpretation applied to the interpretation of the input state will give the interpretation of the output state and the interpretation of the result. (There is a special case when the returned value is a reference, since a references is directly interpreted as its value in the functional world; this is possible since it is necessarily a new reference, according to the typing rules.)

In the following, if θ is a state and ρ (resp. ω) a set of references $\{x_1, \ldots, x_n\}$, then $\theta(\rho)$ (resp. $\theta(\omega)$) denotes the record $\{x_1 = \theta(x_1); \ldots; x_n = \theta(x_n)\}$.

$$
\begin{array}{rcl}
\mathcal{E}(\forall x : t_1.\ t_2) & \stackrel{\text{def}}{=} & \forall x : \mathcal{E}(t_1).\ \mathcal{E}(t_2) \text{ if } t_1 \text{ informative;} \quad \mathcal{E}(t_2) \text{ otherwise} \\
\mathcal{E}(\lambda x : t_1.\ t_2) & \stackrel{\text{def}}{=} & \lambda x : \mathcal{E}(t_1).\ \mathcal{E}(t_2) \text{ if } t_1 \text{ informative;} \quad \mathcal{E}(t_2) \text{ otherwise} \\
\mathcal{E}((t_1\ t_2)) & \stackrel{\text{def}}{=} & (\mathcal{E}(t_1)\ \mathcal{E}(t_2)) \text{ if } t_2 \text{ informative;} \quad \mathcal{E}(t_1) \text{ otherwise} \\
\\
\mathcal{E}(\exists x : t_1.\ t_2) & \stackrel{\text{def}}{=} & \exists x : \mathcal{E}(t_1).\ \mathcal{E}(t_2) \text{ if } t_2 \text{ informative;} \quad \mathcal{E}(t_1) \text{ otherwise} \\
\mathcal{E}((u_1, u_2)) & \stackrel{\text{def}}{=} & (\mathcal{E}(u_1), \mathcal{E}(u_2)) \text{ if } u_2 \text{ informative;} \quad \mathcal{E}(u_1) \text{ otherwise} \\
\mathcal{E}(\mathsf{let}\ (x,y) = u_1\ \mathsf{in}\ u_2) & \stackrel{\text{def}}{=} & \left\{ \begin{array}{ll} \mathsf{let}\ (x,y) = \mathcal{E}(u_1)\ \mathsf{in}\ \mathcal{E}(u_2) & \text{if } u_1 : \exists x : t_1.\ t_2, \\ & \text{with } t_2 \text{ informative;} \\ \mathsf{let}\ x = \mathcal{E}(u_1)\ \mathsf{in}\ \mathcal{E}(u_2) & \text{otherwise} \end{array} \right. \\
\\
\mathcal{E}(\{x_i : t_i\}) & \stackrel{\text{def}}{=} & \{x_i : \mathcal{E}(t_i)\} \\
\mathcal{E}(\{x_i = u_i\}) & \stackrel{\text{def}}{=} & \{x_i = \mathcal{E}(u_i)\} \\
\\
\mathcal{E}(t) & \stackrel{\text{def}}{=} & t, \text{ otherwise}
\end{array}
$$

Fig. 4. *Definition of the extraction operator*

Theorem 1 (*correctness of the functional interpretation*)
Let Γ be a well formed environment whose references are $x_1 : \tau_1\ \mathsf{ref}, \dots, x_n : \tau_n\ \mathsf{ref}$ and e a program such that $\Gamma \vdash_a e : (\tau, (\rho, \omega), \_, \_)$. Let v_i be values of types τ_i and θ the state mapping the x_i to the v_i. Then[5]

$$
\forall \theta', v. \quad \theta.e \longmapsto^\star \theta'.v \quad \Longrightarrow \quad (\overline{e}\ \overline{\theta(\rho)}) = \left\{ \begin{array}{ll} (\overline{\theta'(\omega)}, \overline{\theta'(v)}) & \text{if } \tau = \tau'\ \mathsf{ref} \\ (\overline{\theta'(\omega)}, \overline{v}) & \text{otherwise} \end{array} \right.
$$

where equality stands here and in the following for the conversion $\stackrel{\text{cic}}{=}$ in the Cic (defined in appendix).

Proof
The proof is by induction over the length of the derivation $\longmapsto^\star$ and by case analysis on e. Let us detail the proof for the case $e = \mathsf{ref}\ e_1$. If E stands for the evaluation context $\mathsf{ref}\ []$, the evaluation of e is

$$
\begin{array}{rlr}
\theta.E[e_1] & \longmapsto^\star \theta_1.E[v_1] & \text{with } \theta.e_1 \longmapsto^\star \theta_1.v_1 \\
& \longmapsto \theta_1.\{(x, v_1)\}.x & (\mathit{ref}) \\
& \longmapsto \underbrace{\theta_1 \uplus \{(x, v_1)\}}_{=\theta'}.\underbrace{x}_{=v} & (\theta_{merge})
\end{array}
$$

Since $\theta.e_1 \longmapsto^\star \theta_1.v_1$, and since the effect of e_1 is (ρ, ω) by rule (REF_a), we have by induction hypothesis

$$
(\overline{e_1}\ \overline{\theta(\rho)}) = (\overline{\theta_1(\omega)}, \overline{v_1})
$$

[5] In the expression $\theta.e$, e is to be understood as its interpretation in the formal semantics language, as given in page 730.

Then, by definition of $\overline{e}$, we have

$$\begin{array}{rcl}(\overline{e}\ \overline{\theta(\rho)}) & = & (\overline{e_1}\ \overline{\theta(\rho)}) \\ & = & (\overline{\theta_1(\omega)}, \overline{v_1}) \\ & = & (\overline{\theta'(\omega)}, \overline{\theta'(v)})\end{array}$$

The proofs of the other constructs require several lemmas. The first one expresses that the set ω is really what it pretends to be, that is, that it contains at least all the variables modified by e. This lemma is the following:

Lemma 1
Under the hypotheses of Theorem 1, we have

$$\forall \theta', v. \quad \theta.e \longmapsto^{\star} \theta'.v \quad \Longrightarrow \quad \forall x \in \mathit{dom}(\theta) \backslash \omega.\ \theta'(x) = \theta(x)$$

Another lemma establishes the absence of aliasing in the programs. This property is stated as follows:

Lemma 2
Under the hypotheses of Theorem 1, and when τ is the type of a reference, i.e. $\tau = \tau_1\ \mathsf{ref}$, then v is necessarily a variable and we have

$$v \notin \mathit{dom}(\theta)$$

These two lemmas are proved by induction over the length of $\longmapsto^{\star}$ and by case on e. At last, the proofs for the let in construct and for the function call require the following substitution lemma:

Lemma 3 (*substitution lemma*)
Let Γ be a well formed environment and e a program well typed in Γ. If $z : \tau \in \Gamma$ with $\tau \neq \_\ \mathsf{ref}$ and if v is a value of type τ, then

$$\overline{e[z \leftarrow v]} = \overline{e}[z \leftarrow \overline{v}]$$

The proof is by induction over the structure of e.

The detailed proof of this theorem, and of the lemmas above, are given in (Filliâtre, 1999b), pages 32–35 and 157–162. □

The second theorem expresses that, conversely, if the functional interpretation $\overline{e}$ evaluates on some input store to some output store and some value, then the imperative program e should also evaluate (i.e., should also terminate).

Theorem 2 (*converse of Theorem 1*)
Let Γ be a well formed environment whose references are $x_1 : \tau_1\ \mathsf{ref}, \ldots, x_n : \tau_n\ \mathsf{ref}$ and e a program such that $\Gamma \vdash_a e : (\tau, (\rho, \omega), \_, \_)$. Then for any state θ mapping the x_i to values, we have

$$\forall y, v_0. \quad (\overline{e}\ \overline{\theta(\rho)}) = (y, v_0) \quad \Longrightarrow \quad \exists \theta', v.\ \theta.e \longmapsto^{\star} \theta'.v$$

Proof
To establish this result, we show that the reduction of $\overline{e}$ requires at least as many steps as the evaluation of e. More precisely, if the reduction of $\theta.e$ by $\longmapsto$ requires N steps, then the evaluation of $(\overline{e}\ \overline{\theta(\rho)})$ requires at least N elementary reductions (β-reduction or field access in a record). The proof is by induction over N and by case analysis on the structure of e. The reductions handling the state, namely (θ_{merge}) and (θ_{lift}), are not taken into account (but it is clear that an infinite evaluation of e involves an infinite number of reductions out of those two). The detailed proof is given in (Filliâtre, 1999b), pages 162–165. □

The meaning of extraction is contained in the property of *realizability* which expresses that the extraction of a proof of t is a program satisfying the specification t. We restrict the notion of realizability to the case of propositions $\widehat{\kappa}$; in particular, the informative objects appearing in logical propositions always live in base types (which corresponds to the assumption we made on annotations). Then the notion of realizability can be formally defined as follows:

Definition 10 (*functional realizability, C. Paulin (Paulin-Mohring, 1989b)*)
The proposition $x\ \mathsf{r}_f\ t$ is defined recursively over the informative proposition t in the following way:

$$
\begin{array}{lcl}
x\ \mathsf{r}_f\ \forall y : t_1.\, t_2 & \stackrel{\text{def}}{=} & \forall y : \mathcal{E}(t_1).\ y\ \mathsf{r}_f\ t_1 \Rightarrow (x\ y)\ \mathsf{r}_f\ t_2 \\
x\ \mathsf{r}_f\ t_1 \rightarrow t_2 & \stackrel{\text{def}}{=} & t_1 \Rightarrow x\ \mathsf{r}_f\ t_2 \\
x\ \mathsf{r}_f\ \exists y : t_1.\, t_2 & \stackrel{\text{def}}{=} & x\ \mathsf{r}_f\ t_1\ \wedge\ t_2[y \leftarrow x] \\
x\ \mathsf{r}_f\ \{x_i : t_i\} & \stackrel{\text{def}}{=} & \forall i.\ x.x_i\ \mathsf{r}_f\ t_i \\
x\ \mathsf{r}_f\ t & \stackrel{\text{def}}{=} & \mathsf{True},\ \text{otherwise.}
\end{array}
$$

This definition comes with the following realizability theorem (see (Paulin-Mohring, 1989b), page 85).

Theorem 3 (*realizability theorem, C. Paulin (Paulin-Mohring, 1989b)*)
Let Γ be a proof environment in the Calculus of Inductive Constructions. Then

$$\Gamma \vdash_{\text{CIC}} u : t \quad \Longrightarrow \quad \mathcal{E}(u)\ \mathsf{r}_f\ t$$

4.3 Correctness result

Finally, we have to define the notion of program correctness with respect to specifications. The usual definition of total correctness for a given Hoare triplet $\{P\}\ e\ \{Q\}$ is a formalization of the sentence "In any state satisfying P, the execution of e terminates and the resulting state satisfies Q". In our case, the result also has to be taken into account, and since it may be a function, we have to express that it satisfies its own specification. Similarly, functions may be present in the environment, and in particular inside references, and we have to express that the corresponding values satisfy the declared specifications.

Inspired by the notion of functional realizability, we define the notion of correctness corresponding to our programs as "imperative realizability".

Definition 11 (*imperative realizability*)
Let Γ be a well formed environment and τ and κ some types of values and computations such that $\Gamma \vdash_a \tau\ \mathsf{wf}$ and $\Gamma \vdash_a \kappa\ \mathsf{wf}$. Then for any value v of type τ and any program e of type κ the propositions $e\ \mathsf{r}_i\ \kappa$ and $v\ \mathsf{r}_i\ \tau$ are mutually recursively defined as follows:

$$\begin{array}{lcl}
e\ \mathsf{r}_i\ (r:\tau,(\rho,\omega),P,Q) & \stackrel{\text{def}}{=} & \forall\theta.\ \theta\ \mathsf{r}_i\ \rho \Rightarrow P(\theta(\rho)) \Rightarrow \exists\theta',v.\ \theta.e \longmapsto^{\star} \theta'.v \\
 & & \quad \wedge\, \theta'\ \mathsf{r}_i\ \omega \wedge v\ \mathsf{r}_i\ \tau \wedge Q(\theta(\rho),\theta'(\omega),v) \\
e\ \mathsf{r}_i\ (r:\tau\ \mathsf{ref},(\rho,\omega),P,Q) & \stackrel{\text{def}}{=} & \forall\theta.\ \theta\ \mathsf{r}_i\ \rho \Rightarrow P(\theta(\rho)) \Rightarrow \exists\theta',x.\ \theta.e \longmapsto^{\star} \theta'.x \\
 & & \quad \wedge\, \theta'\ \mathsf{r}_i\ \omega \wedge \theta'(x)\ \mathsf{r}_i\ \tau \wedge Q(\theta(\rho),\theta'(\omega),\theta'(x)) \\
 & & \\
v\ \mathsf{r}_i\ \beta & \stackrel{\text{def}}{=} & \mathsf{True} \\
v\ \mathsf{r}_i\ (x:\tau\ \mathsf{ref}) \rightarrow \kappa & \stackrel{\text{def}}{=} & \forall r:\tau\ \mathsf{ref}.\ (v\ r)\ \mathsf{r}_i\ \kappa[x \leftarrow r] \\
v\ \mathsf{r}_i\ (x:\tau) \rightarrow \kappa & \stackrel{\text{def}}{=} & \forall x:\tau.\ x\ \mathsf{r}_i\ \tau \Rightarrow (v\ x)\ \mathsf{r}_i\ \kappa
\end{array}$$

with $\theta\ \mathsf{r}_i\ \{x_1,\dots,x_n\} \stackrel{\text{def}}{=} \forall i \in \{1,\dots,n\}.\ x_i : \tau_i\ \mathsf{ref} \in \Gamma \wedge \theta(x_i)\ \mathsf{r}_i\ \tau_i$

The correctness of a program e expresses that in any state θ which values have expected types ($\theta\ \mathsf{r}_i\ \rho$) and satisfy the precondition P, the program e evaluates to a state θ' and a value v such that the values in θ' have expected types ($\theta'\ \mathsf{r}_i\ \omega$), v satisfies its specification ($v\ \mathsf{r}_i\ \tau$) and the postcondition Q is satisfied. The definition is similar when the result is a reference. Regarding values, the definition resembles the one for functional realizability. With respect to the traditional ways of expressing the correctness of imperative programs, there is here the additional difficulty of having functions as first-class values, thus possibly in references and as results of computations.

We are now in position to prove the correctness of our method, that is, that a complete proof of $\widehat{e}$ implies $e\ \mathsf{r}_i\ \kappa$, for any program e of type κ. This is related to the fact that $\overline{e}$ respects the semantics of e. The key lemma is the following, which relates the imperative and functional notions of realizability.

Lemma 4
Let Γ be a well formed environment, κ a well formed type in Γ and e a program of type κ in Γ. Then

$$\overline{e}\ \mathsf{r}_f\ \widehat{\kappa} \quad\Longleftrightarrow\quad e\ \mathsf{r}_i\ \kappa$$

Proof
We show by mutual induction over the structure of types that for any type of value τ and any *value* v of type τ we have

$$\overline{v}\ \mathsf{r}_f\ \widehat{\tau} \quad\Longleftrightarrow\quad v\ \mathsf{r}_i\ \tau$$

and that for any type of computation e and any program e of type κ we have

$$\overline{e}\ \mathsf{r}_f\ \widehat{\kappa} \quad\Longleftrightarrow\quad e\ \mathsf{r}_i\ \kappa$$

The cases of values (base types, type of a reference and function type) are almost immediate (using the substitution lemma above for the case of a function taking a reference as argument). The key point is the case of a computation.

Let $\kappa = (r : \tau, (\rho, \omega), P, Q)$, with $\tau \neq \_\ \mathsf{ref}$. We assume $\overline{e}\ \mathsf{r}_f\ \widehat{\kappa}$, that is,

$$\forall x : \overline{\rho}.\ x\ \mathsf{r}_f\ \widehat{\rho} \Rightarrow P(x) \Rightarrow \mathsf{let}\ (y, r) = (\overline{e}\ x)\ \mathsf{in}\ y\ \mathsf{r}_f\ \widehat{\omega} \wedge r\ \mathsf{r}_f\ \widehat{\tau} \wedge Q(x, y, r) \tag{5}$$

and we want to show that $e\ \mathsf{r}_i\ \kappa$, that is,

$$\begin{aligned} &\forall \theta.\ \theta\ \mathsf{r}_i\ \rho \Rightarrow P(\theta(\rho)) \Rightarrow \exists \theta', v. \\ &\qquad \theta.e \longmapsto^\star \theta'.v \wedge \theta'\ \mathsf{r}_i\ \omega \wedge v\ \mathsf{r}_i\ \tau \wedge Q(\theta(\rho), \theta'(\omega), v) \end{aligned} \tag{6}$$

Let θ be a state such that $\theta\ \mathsf{r}_i\ \rho$ and $P(\theta(\rho))$. Let $x = \overline{\theta(\rho)}$, that is, $\mathcal{E}(\widehat{\theta(\rho)})$. We have $\theta\ \mathsf{r}_i\ \rho$ and thus by induction hypothesis applied to each field of x we have $x\ \mathsf{r}_f\ \widehat{\rho}$. We also have $P(x)$, since P only mentions objects of x of base types, which are therefore equal to their computational contents. So the hypotheses of (5) are satisfied. Then let $(y, r) = (\overline{e}\ x)$. By Theorems 1 and 2, we have

$$\theta.e \longmapsto^\star \theta'.v \quad \text{with } y = \overline{\theta'(\omega)} \text{ and } r = \overline{v}$$

We have $r\ \mathsf{r}_f\ \widehat{\tau}$ and thus $v\ \mathsf{r}_i\ \tau$ by induction hypothesis. Similarly, we have $y\ \mathsf{r}_f\ \widehat{\omega}$ and thus $\theta'\ \mathsf{r}_i\ \omega$, applying the induction hypothesis on each member. Last, we have $Q(x, y, r)$ and thus $Q(\theta(\rho), \theta'(\omega), v)$ (remember that, as P does, Q only mentions objects of base types; in particular, Q may mention r only if r lives in a base type and in this case we have $r = \overline{v} = v$). So (6) is established, that is, $e\ \mathsf{r}_i\ \kappa$.

Conversely, let us assume (6) and let us prove (5). Let x be a record of type $\overline{\rho}$ such that $x\ \mathsf{r}_f\ \widehat{\rho}$ and $P(x)$. Let θ be a state such that $\overline{\theta(\rho)} = x$. We have $x\ \mathsf{r}_f\ \widehat{\rho}$ and thus $\theta\ \mathsf{r}_i\ \rho$ by induction hypothesis applied on each field of x. We have $P(\theta(\rho))$, for the same reason as in the converse part. Then from (6) there exists θ' and v such that $\theta.e \longmapsto^\star \theta'.v$. By Theorem 1, we have $(\overline{e}\ x) = (\overline{\theta'(\omega)}, \overline{v})$. Let us write (y, r) this tuple. By (6), we have $\theta'\ \mathsf{r}_i\ \omega$ and thus $y\ \mathsf{r}_f\ \widehat{\omega}$ by induction hypothesis applied on each field of y. Similarly, we have $v\ \mathsf{r}_i\ \tau$ from (6) and thus $r\ \mathsf{r}_f\ \widehat{\tau}$ by induction hypothesis. At last, we have $Q(\theta(\rho), \theta'(\omega), v)$ from (6) and thus $Q(x, y, r)$ for the same reason as in the converse part. So we have established (5), that is, $\overline{e}\ \mathsf{r}_f\ \widehat{\kappa}$. □

Then we can state and prove the main result.

Theorem 4 (*correctness*)
Let Γ be a well formed environment and e a program such that $\Gamma \vdash_a e : \kappa$. If all the obligations in $\widehat{e}$ can be replaced by proof terms, in such a way that $\widehat{e}$ becomes a proof of $\widehat{\kappa}$, then the proposition $e\ \mathsf{r}_i\ \kappa$ holds.

Proof
If $\widehat{e}$ is a proof of $\widehat{\kappa}$ then its informative contents realizes, in the functional meaning, the proposition $\widehat{\kappa}$. Since the informative contents of $\widehat{e}$ is exactly $\overline{e}$, by definition, we have $\overline{e}\ \mathsf{r}_f\ \widehat{\kappa}$ by the realizability theorem. Then by Lemma 4, we have $e\ \mathsf{r}_i\ \kappa$. □

5 Implementation and case studies

This work has been implemented in the Coq proof assistant (Coq, 2001) and is already released with the system. It is fully documented in the Coq reference manual. In this section, we will give an overview of the implementation, discuss some completeness issues and describe some case studies.

5.1 Implementation in the system Coq

The implementation consists of 6000 lines of Objective Caml code. It appears as a tactic called Correctness taking an annotated program as argument and generating a set of goals, which are logical propositions to be proved by the user. Given an annotated program e, the tactic Correctness applies the following steps:

1. It determines the type of computation κ of e by the typing algorithm.
2. The proposition $\widehat{\kappa}$ is computed and declared as a goal.
3. The partial proof term $\widehat{e}$ is computed following Definition 8 and is given to the proof engine, using the Refine tactic developed on purpose, and each hole in $\widehat{e}$ leads to a subgoal.
4. Once the proofs are completed, the program is added to the environment and may be used in other programs.

It is also possible to only declare programs in the environment, so that one can assume some procedures and use them before implementing them. The extraction mechanism of Coq is extended in order to extract the imperative programs as such in the target languages of extraction which have imperative features.

Our implementation leads to several improvements with respect to what has been presented in this paper. First, the language includes arrays of purely functional values. They can be seen as references on purely functional finite mappings. This means that $t[e]$ may be read as let $n = e$ in $(access\ !t\ n)$, and $t[e_1] := e_2$ as let $n = e_1$ in let $v = e_2$ in $t := (store\ !t\ n\ v)$, where *access* and *store* are the operations of the purely functional finite mappings. Of course, arrays are extracted as real arrays when the code is produced. Second, some features have been added to simplify the specification of programs—mainly, the possibility of inserting labels in the programs and referring in annotations to the value that a reference had at the program points corresponding to those labels. This is easy to implement since our interpretation defines several variables for the successive values of a reference. Consequently, the auxiliary variables are no more useful, and the rewriting reasoning they imply is avoided. In particular, a loop invariant may mention the values of the references at some point before the loop using such a label.

5.2 Completeness issues

The completeness expresses the converse of Theorem 4, that is, that the proof obligations are provable as soon as the program satisfies the specification. Although the correctness seems to be a much more important property, the completeness is far from being a theoretical issue. We can illustrate it with a small example. Let us consider the following piece of code:

```
begin
  if x < y then m := x else m := y;
  m := !m - 1
end
{ m < x }
```

It obviously satisfies its specification. Let us look at the proof obligations generated for this program. Its interpretation as a partial proof term is

$$\begin{array}{ll} \mathsf{let}\ (\{m = m_1\}, \_) = & \mathsf{let}\ (b, q) = \widehat{x < y}\ \mathsf{in} \\ & \mathsf{if}\ b\ \mathsf{then}\ (\{m = x\}, ())\ \mathsf{else}\ (\{m = y\}, ())\ \ \mathsf{in} \\ \multicolumn{2}{l}{\mathsf{let}\ (\{m = m_2\}, v) = (\{m = m_1 - 1\}, ())\ \mathsf{in}} \\ \multicolumn{2}{l}{(\{m = m_2\}, v, ?)} \end{array}$$

The second let is a redex, and hence can be eliminated. But the first one is not, and consequently we get the following proof obligation:

$$\frac{m_1 \quad : \quad Z}{m_1 - 1 < x}$$

which cannot be proved, since we do not have enough information about the variable m_1. This variable stands for the value of m after the first assignment. The only way to obtain a provable obligation is to get m_1 together with a property about it. This can be achieved by annotating the conditional with a postcondition, in the following way:

1. begin
2. (if $x < y$ then $m := x$ else $m := y$) $\{ m \leq x \}$;
3. $m :=\ !m - 1$
4. end
5. $\{ m < x \}$

Then we will get, in addition to the two obligations establishing this new annotation, the following provable obligation:

$$\frac{\begin{array}{lll} m_1 & : & Z \\ q_1 & : & m_1 \leq x \end{array}}{m_1 - 1 < x}$$

The inserted annotation $m \leq x$ may seem quite *ad hoc*. Actually, there was a general solution, which consisted in inserting the *weakest precondition* computed from the postcondition $m < x$ and the assignment $m :=\ !m - 1$, that is $m - 1 < x$.

More generally, we have shown in (Filliâtre, 1999b) that, for a fragment of our language, there exists a canonical way to annotate a program using weakest preconditions that leads to provable obligations as soon as the program satisfies its specification. This has been integrated in the implementation in order to release the user from inserting too many annotations. Notice that it does not imply more proof obligations. Indeed, if the program fragment $e_1\ ;\ e_2\ \{Q\}$, for instance, is given an intermediate annotation computed as the weakest precondition of e_2 and Q, that is,

$$e_1\ \{\mathrm{wp}(e_2)(Q)\}\ ;\ e_2\ \{Q\}$$

then the second proof obligation will be automatically discharged, since it is an immediate consequence of the proof of $\mathrm{wp}(e_2)(Q)$ and of the substitutions introduced in Q by e_2.

Therefore, the only annotations that must be inserted into the programs are the loop invariants and the functions' pre- and postconditions, as it has been confirmed by all the case studies.

5.3 Case studies

Some nontrivial algorithms have been certified by the author using the tactic Correctness:

- Three in place sorting algorithms: insertion sort, quicksort and heapsort
- Knuth-Morris-Pratt fast string searching algorithm
- The program *Find* which had been proved correct by C. A. R. Hoare in (Hoare, 1971)

Those case studies are detailed in (Filliâtre, 1999b) and available on the Coq web page (Coq, 2001). The proofs of correctness of the sorting algorithms are presented in (Filliâtre & Magaud, 1999). Our formal proof of the program *Find* is presented in (Filliâtre, 2001); in particular, we show that we obtain exactly the same proof obligations as Hoare in his paper. We are not going to describe these case studies here, since the interested reader may consult (Filliâtre, 1999b; Filliâtre, 2001; Filliâtre & Magaud, 1999) for fully detailed descriptions. However, it is interesting to have a look at the characteristic metrics of these developments, which are summarized in Table 1.

For each development, we give some informations about the relative sizes of the code, the specification and the proof, the total development time and the time needed to recheck the proof. First, we notice that the specification of a complex algorithm is not so difficult: only a few lines of definitions are necessary and the number of annotations is always between the third and the half of the total number of code lines. Second, the proof itself is not so big. The number of proof obligations is similar to the number of code lines. The number of proof steps needed to discharge the proof obligations may seem important to a reader not familiar with Coq's proofs, but the total amount of development time is speaking by itself: three days to establish the correctness of an algorithm as complex as inplace heapsort is clearly the sign of a sane and efficient methodology.

6 Discussion

We have proposed a method to certify programs combining functional and imperative features in the context of type theory. This method benefits from the power of type theory as a specification language and from the existing proof assistants as proof tools. Our method is based on a logical interpretation of annotated programs as partial proof terms of their specifications. One of the main interests is a direct interpretation of functional features, usually painfully handled in other frameworks. Our interpretation is based on a static analysis of the programs' effects and on a classic monadic call-by-value translation.

Table 1. *Some metrics about the case studies*

	Find	**Quicksort**	**KMP**	**Heapsort**
Specification				
Lines of specification	29	13	17	23
Lines of code (# functions)	27 (1)	41 (4)	24 (2)	19 (2)
Annotations	9	18	8	7
Proof				
Proof obligations	22	26	23	22
Lemmas (manually introduced)	39	11	11	28
Proof steps	619	468	377	626
Total development time	—	2 days	1,5 days	3 days
Compilation time[a]	9 mn 02 s	8 mn 06 s	5 mn 45 s	12 mn 07 s

[a] Compilations were realized on a Pentium 200 Mhz 64 MB RAM running Linux.

Future work. We expect our interpretation to be easily extended to other programming features (exceptions, input-output and so forth) by an adequate extension of the notion of effect and of the corresponding interpretation. The treatment of exceptions is particularly important if we want to deal with realistic ML programs. Ultimately, we would like to unify the work of C. Parent (Parent, 1995) and ours on the functional fragment of our language.

To allow the certification of large programs, our method should also provide an appropriate notion of *modularity*. Although there is still a lot to do here, one can notice that our method is already modular: indeed, our notion of annotated type, containing the traditional type, an effect and a specification, is exactly what is required in using a program. (Actually, it is already possible to declare a program and use it, without giving it any implementation.) For the same reason, it would be possible to process by *refinements*, by allowing holes in programs whose types of computation would be given and which would be refined later. We plan to implement both modularity and refinement in the next version of our implementation.

Polymorphism. In this paper, we do not consider polymorphism, although it is directly available in the Calculus of Inductive Constructions, because it would require an effect polymorphism, that is, the possibility to generalize the type of a function with respect to an effect. Effect inference is still possible in this case, as is done in *The Type and Effect Discipline* (Talpin & Jouvelot, 1994), but we would not be able to define the interpretations of types and programs anymore: if a function f has type $\forall\alpha.\forall\epsilon.\alpha \rightarrow^{\epsilon} \alpha$, the type of its interpretation depends on ϵ and is not computable statically. It must be made *explicit* as a function over ϵ. Although it would be possible to *internalize* the definitions of the interpretations $\widehat{\tau}$ and $\widehat{\kappa}$, there is a practical price to pay. Moreover, polymorphic functions in programming languages with imperative features are really difficult to specify. Consider, for instance, the problem of specifying the polymorphic function map such that $(\mathsf{map}\ f\ [v_1; v_2; \ldots; v_n]) = [(f\ v_1); (f\ v_2); \ldots; (f\ v_n)]$. In a purely functional framework, one can state properties such as $(\mathsf{map}\ f\ (v :: l)) = (f\ v) :: (\mathsf{map}\ f\ l)$ or $(\mathsf{nth}\ (\mathsf{map}\ f\ l)\ i) = (f\ (\mathsf{nth}\ l\ i))$, where the equality is the mathematical one. But when f may have arbitrary side effects, this is no longer meaningful. In particular, the order in which the $(f\ v_i)$'s are computed is relevant.

In (Filliâtre, 1999b), we consider a simple notion of polymorphism, where types can be generalized with respect to type variables but not to effects.

Related work. The type and effect discipline has been introduced by J.-P. Talpin and P. Jouvelot (Talpin & Jouvelot, 1994), and followed by several other systems (Wright, 1992; Jouvelot & Gifford, 1991); in particular, effects systems for exceptions were derived from this work (Guzmán & Suárez, 1994; Leroy & Pessaux, 2000), which could be adapted in our framework to add exceptions to our language. Although we do not use typing systems as powerful as the previous ones, our contribution is more in the use of typing to exclude aliases, which is the key to a natural functional interpretation.

Our interpretation of imperative programs in a functional world is inspired by the call-by-value translations introduced by E. Moggi (Moggi, 1991) and P. Wadler (Wadler, 1993). In (Filliâtre, 1999b), we proposed a notion of monads parameterized with effects which generalizes what is presented in this paper. Similar combinations of effects and monads, in the context of more powerful effects systems, have been studied by P. Wadler (Wadler, 1998), A. Tolmach (Tolmach, 1998), M. Semmelroth and A. Sabry (Semmelroth & Sabry, 1999), and E. Moggi and F. Palumbo (Moggi & Palumbo, 1999).

Semantics for reasoning about LISP-like programs with side effects have been proposed by I. Mason, C. Talcott and others (Mason & Talcott, 1989; Honsell *et al.*, 1992) but the focus is mainly on operational equivalence. Our purpose was rather to advocate in favor of Hoare triples where mathematical variables stand directly for the values contained in references.

Comparison with other methods. We can compare our method to some other formal methods, on the different point of view of the specification, the programs one can write and the proofs. Among the existing formal methods, we can compare our work to VDM (the Vienna Development Method) (Jones, 1980; Jones, 1989), J. R. Abrial's B method (Abrial, 1996) and the system KIV (Karlsruhe Interactive Verifier) (Reif, 1995), which are three very different approaches to formal verification.

From the point of view of specification, the VDM and B methods are based on an axiomatization of first-order set theory. Those are heavy formalizations, with numerous axioms, and whose expressivity is difficult to figure out. A set-theoretical approach implies additional difficulties, such as the definition of new data types or proof obligations of the kind $n \in NAT$. In our case, the choice of the Calculus of Inductive Constructions is a satisfactory solution on both theoretical and practical grounds. Independently of the logical framework, our approach to specification is close to VDM's: programs are specified using pre- and postconditions, and programs' effects must be explicitly declared as sets of read and written variables. With the B method, on the other hand, the operations of an abstract machine are specified using generalized substitutions, which can be seen as an extension of E. W. Dijkstra's guarded commands. It is a rather different approach to the specification problem, but it has been proved that it is equivalent to the use of pre- and postconditions (see (Abrial, 1996), pages 292–295). In the system KIV, programs are given *algebraic specifications*, that are composed of signatures and sets of equational axioms. Those specifications mention only purely functional objects, which are explicitly related later to the implementations. In some sense, there is a similarity between the purely functional interfaces of the system KIV and our functional interpretations of programs.

Regarding specifications, our programming experience leads us to think that pre- and postconditions are the most natural way to specify programs, whether functional or imperative. Then the choice of the Calculus of Inductive Constructions affords mathematical simplicity and expressivity, which is needed for complex programs. For instance, when doing the correctness proof of the *heapsort* algorithm, we easily defined a *heap* predicate as an inductive property and reasoned by induction over this predicate in most lemmas. This would have been much more complicated within a first-order set-theoretical framework.

From the point of view of the programming language, formal methods provide only a small set of semantically well understood constructs. Those are usually local variables, assignments, sequences, conditionals, loops, and nothing else. In the methods VDM, B and KIV, procedures, functions and recursive functions are not first-class values, but top-level objects. Our ML background naturally led us to consider a larger set of program constructs. References are really first-class values, which can be dynamically created and returned as results of functions. Functions are also first-class values, which can be passed as arguments and returned as results. This was possible because type theory naturally handles functional objects. Finally, our approach does not set the data types once and for all, but rather assumes them to be defined in the underlying type theory. This is an easy solution in practice, but this is not adapted to real world programs where, for instance, one has to cope with overflows when using machine integers.

The last point of comparison is the proof tool. A formal method is incomplete without a good proof tool to establish the validity of the proof obligations. Regarding VDM, the attempts to define a proof tool were not very successful, and VDM is therefore a development methodology rather than a certification tool. The B method, which also relies on a set-theoretic framework, is equipped with several good provers, whose automation is impressive (more than 80% of proof obligations automatically discharged for real code). The system KIV also provides impressive figures regarding automatic proofs, although there is no description of the prover and its theoretical foundations. On the contrary, the Coq proof assistant is not equipped with powerful automatic tactics but focuses rather on interactive proofs. Above all, the Coq system provides a really safe proof assistant, being based on a small set of logical rules. (The consistency of the Calculus of Inductive Constructions has been formally proved by B. Barras (Barras, 1999).)

Nevertheless, our method has not yet reached the maturity of methods like B or KIV. Indeed, such methods provide modular development mechanisms. Moreover, the B method also provides a stepwise refinement mechanism, which allows programs and specifications to be built step by step, proof obligations being generated at each step. These features are still only prospectives of our work.

Acknowledgments. We thank the anonymous referees for their detailed and helpful comments.

References

Abrial, J. R. (1996). *The B-Book. Assigning programs to meaning.* Cambridge University Press.

Back, R. J. R. (1981). On correct refinement of programs. *Journal of Computer and Systems Sciences*, **23**(1), 49–68.

Barras, B. 1999 (Nov.). *Auto-validation d'un système de preuves avec familles inductives.* Thèse de doctorat, Université Paris 7.

Coq. (2001). *The Coq Proof Assistant.* `http://coq.inria.fr/`.

Coquand, T., & Huet, G. (1988). The Calculus of Constructions. *Information and computation*, **76**(2/3), 95–120.

Cousot, P. (1990). *Handbook of Theoretical Computer Science.* Vol. B. Elsevier Science Publishers B. V. Chap. Methods and Logics for Proving Programs, pages 841–993.

de Bruijn, N.J. (1980). A survey of the project Automath. Seldin, J. P., & Hindley, J. R. (eds), *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.* Academic Press.

Dijkstra, E. W. (1976). *A Discipline of Programming.* Prentice Hall.

Filliâtre, J.-C. 1999a (November). *A theory of monads parameterized by effects.* Research Report 1367. LRI, Université Paris Sud.

Filliâtre, J.-C. 1999b (July). *Preuve de programmes impératifs en théorie des types.* Thèse de doctorat, Université Paris-Sud.

Filliâtre, J.-C. (2001). Formal Proof of a Program: Find. *Science of computer programming.* To appear.

Filliâtre, J.-C., & Magaud, N. (1999). Certification of sorting algorithms in the system Coq. *Theorem proving in higher order logics: Emerging trends.*

Floyd, R. W. (1967). Assigning meanings to programs. *Pages 19–32 of:* Schwartz, J. T. (ed), *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics 19.* American Mathematical Society, Providence.

Girard, J.-Y. (1972). *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* Thèse de doctorat d'État, Univerité Paris 7.

Guzmán, J., & Suárez, A. 1994 (June). An Extended Type System for Exceptions. *Record of the fifth ACM SIGPLAN workshop on ML and its Applications.* Also appears as Research Report 2265, INRIA, BP 105 - 78153 Le Chesnay Cedex, France.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, **12**(10), 576–580,583. Also in (Hoare & Jones, 1989) pages 45–58.

Hoare, C. A. R. (1971). Proof of a program: *Find. Communications of the ACM*, **14**(1), 39–45. Also in (Hoare & Jones, 1989) pages 59–74.

Hoare, C. A. R., & Jones, C. B. (1989). *Essays in Computing Science.* Prentice Hall.

Honsell, F., Mason, I. A., Smith, S. F., & Talcott, C. L. (1992). A Theory of Classes for a Functional Language with Effects. *Pages 309–326 of: 1992 Annual Conference of the European Association for Computer Science Logic CSL92, San Miniato.* Lecture Notes in Computer Science, vol. 702. Springer-Verlag.

Jones, C. B. (1980). *Software Development. A rigorous approach.* Prentice Hall.

Jones, C. B. (1989). *Systematic Software Development Using VDM.* Prentice Hall.

Jouvelot, P., & Gifford, D. K. (1991). Algebraic Reconstruction of Types and Effects. *Pages 303–310 of: Proceedings of the 18th ACM Symposium on Principles of Programming Languages.* ACM.

Leroy, X., & Pessaux, F. (2000). Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems*, **2**(22).

Mason, I. A., & Talcott, C. L. (1989). Axiomatizing Operational Equivalence in the Presence of Side Effects. *Pages 284–293 of: Fourth Annual Symposium on Logic in Computer Science.* IEEE.

Moggi, E. (1991). Notions of computations and monads. *Information and Computation*, **93**(1).

Moggi, E., & Palumbo, F. (1999). Monadic Encapsulation of Effects: a Revised Approach. *Third International Workshop on Higher Order Operational Techniques in Semantics.*

Morgan, C. C. (1990). *Programming from Specifications.* Prentice Hall.

Parent, C. 1993 (Oct.). *Developing certified programs in the system Coq – The Program tactic.* Tech. rept. 93-29. École Normale Supérieure de Lyon. Also in *Proceedings of the BRA Workshop Types for Proofs and Programs*, May 93.

Parent, C. 1995 (Janvier). *Synthèse de preuves de programmes dans le Calcul des Constructions Inductives.* Thèse de doctorat, École Normale Supérieure de Lyon.

Paulin-Mohring, C. (1989a). Extracting F_ω's programs from proofs in the Calculus of Constructions. *Sixteenth Annual ACM Symposium on Principles of Programming Languages.* Austin: ACM.

Paulin-Mohring, C. 1989b (Janvier). *Extraction de programmes dans le Calcul des Constructions.* Thèse de doctorat, Université de Paris VII.

Paulin-Mohring, C. (1993). Inductive Definitions in the System Coq - Rules and Properties. Bezem, M., & Groote, J.-F. (eds), *Proceedings of the conference Typed Lambda Calculi and Applications.* Lecture Notes in Computer Science, vol. 664. Springer-Verlag. Also LIP research report 92-49.

Reif, W. (1995). The KIV-approach to software verification. Broy, M., & Jähnichen, S. (eds), *KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report.* Lecture Notes in Computer Science, vol. 1009. Springer-Verlag.

Semmelroth, M., & Sabry, A. (1999). Monadic Encapsulation in ML. *ACM SIGPLAN International Conference on Functional Programming.* ACM.

Talpin, J.-P., & Jouvelot, P. (1994). The Type and Effect discipline. *Information and computation*, **111**(2), 245–296.

Tolmach, A. (1998). Optimizing ML Using a Hierarchy of Monadic Types. *Pages 97–113* *of: Types in Compilation '98 Workshop.* Lecture Notes in Computer Science, vol. 1473. Kyoto, Japan: Springer-Verlag.

Wadler, P. (1993). Monads for functional programming. Broy, M. (ed), *Program Design Calculi.* NATO ASI Series. Springer Verlag.

Wadler, P. (1998). The marriage of effects and monads. *Pages 63–74 of: International Conference on Functional Programming.* Baltimore: ACM.

Wright, A. K. (1992). Typing References by Effect Inference. *Pages 473–491 of: European Symposium on Programming.* Lecture Notes in Computer Science, vol. 582. Springer-Verlag.

Wright, A. K., & Felleisen, M. (1994). A syntactic approach to type soundness. *Information and computation*, **115**, 38–94.

Appendix: Typing rules for Cic

Sorts and terms of the Calculus of Constructions are given by the following grammar:

$$\begin{array}{llll} \text{sorts} & s & ::= & \mathsf{Set} \mid \mathsf{Prop} \mid \mathsf{Type}(i) \\ \text{terms} & t & ::= & s \mid c \mid x \mid \forall x : t.\ t \mid \lambda x : t.\ t \mid (t\ t) \end{array}$$

In the following typing rules, the typing judgment $\vdash_{\text{CIC}}$ is simply written as $\vdash$. The conversion $\stackrel{\text{cic}}{=}$ is the reflexive transitive congruent closure of the reduction $\triangleright$ defined below.

Basic Rules for the Calculus of Constructions

$$\frac{s \in \{\mathsf{Set}, \mathsf{Prop}\}}{\Gamma \vdash s : \mathsf{Type}(i)} \qquad \frac{i < j}{\Gamma \vdash \mathsf{Type}(i) : \mathsf{Type}(j)}$$

$$\frac{x : t \in \Gamma}{\Gamma \vdash x : t} \qquad \frac{c : t \in \Gamma}{\Gamma \vdash c : t}$$

$$\frac{\Gamma \vdash t_1 : s_1 \qquad \Gamma, x : t_1 \vdash t_2 : s_2 \qquad s_1, s_2 \in \{\mathsf{Set}, \mathsf{Prop}\}}{\Gamma \vdash \forall x : t_1.\ t_2 : s_2}$$

$$\frac{\Gamma \vdash t_1 : \mathsf{Type}(i) \qquad \Gamma, x : t_1 \vdash t_2 : \mathsf{Type}(j) \qquad i, j \leq k}{\Gamma \vdash \forall x : t_1.\ t_2 : \mathsf{Type}(k)}$$

$$\frac{\Gamma \vdash \forall x : t_1.\ t_2 : s \qquad \Gamma, x : t_1 \vdash u_2 : t_2}{\Gamma \vdash \lambda x : t_1.\ u_2 : \forall x : t_1.\ t_2}$$

$$\frac{\Gamma \vdash u_2 : \forall x : t_1.\ t_2 \qquad \Gamma \vdash u_1 : t_1}{\Gamma \vdash (u_2\ u_1) : t_2[x \leftarrow u_1]}$$

$$(\lambda x : t.u_1\ u_2)\ \triangleright\ u_1[x \leftarrow u_2] \qquad \frac{\Gamma \vdash t_2 : s \qquad \Gamma \vdash u : t_1 \qquad t_1 \stackrel{\text{cic}}{=} t_2}{\Gamma \vdash u : t_2}$$

Existential quantification and records can be defined in the Calculus of Inductive Constructions as inductive types—as done for instance in the Coq proof assistant (Coq, 2001). Though, it is simpler here to consider them as primitive notions, avoiding the complex rules for inductive types and their eliminations.

Existential Quantification

$$t \ ::= \ \ldots \mid \exists x : t.\ t \mid (t,t) \mid \mathsf{let}\ (x,x) = t\ \mathsf{in}\ t$$

$$\frac{\Gamma \vdash t_1 : \mathsf{Set} \qquad \Gamma, x : t_1 \vdash t_2 : s \quad s \in \{\mathsf{Set}, \mathsf{Prop}\}}{\Gamma \vdash \exists x : t_1.\ t_2 : \mathsf{Set}}$$

$$\frac{\Gamma \vdash \exists x : t_1.\ t_2 : \mathsf{Set} \qquad \Gamma \vdash u_1 : t_1 \qquad \Gamma \vdash u_2 : t_2[x \leftarrow u_1]}{\Gamma \vdash (u_1, u_2) : \exists x : t_1.\ t_2}$$

$$\frac{\Gamma \vdash u_1 : \exists x : t_1.\ t_2 \qquad \Gamma, x : t_1, x_1 : t_2 \vdash u_2 : t_3 \qquad x, x_1 \notin FV(t_3)}{\Gamma \vdash \mathsf{let}\ (x, x_1) = u_1\ \mathsf{in}\ u_2 : t_3}$$

$$\mathsf{let}\ (x_1, x_2) = (u_1, u_2)\ \mathsf{in}\ u_3 \ \rhd \ u_3[x_1 \leftarrow u_1; x_2 \leftarrow u_2]$$

Non-Dependent Records

$$t \ ::= \ \ldots \mid \{x : t; \ldots; x : t\} \mid \{x = t; \ldots; x = t\} \mid t.x$$

$$\frac{\Gamma \vdash t_i : \mathsf{Set}}{\Gamma \vdash \{x_i : t_i\} : \mathsf{Set}}$$

$$\frac{\Gamma \vdash t_i : \mathsf{Set} \qquad \Gamma \vdash u_i : t_i}{\Gamma \vdash \{x_i = u_i\} : \{x_i : t_i\}} \qquad\qquad \frac{\Gamma \vdash t : \{x_i : t_i\}}{\Gamma \vdash t.x_i : t_i}$$

$$\{x_i = u_i\}.x_j \ \rhd \ u_j$$