Proof of Imperative Programs in Type Theory

Jean-Christophe Filliâtre*

LRI, URA CNRS 410, Bât. 490, Université Paris Sud,
91405 ORSAY Cedex, FRANCE
Jean-Christophe.Filliatre@lri.fr
www.lri.fr/~filliatr

Abstract. We present a new approach to certifying functional programs with imperative aspects, in the context of Type Theory. The key is a functional translation of imperative programs, based on a combination of the type and effect discipline and monads. Then an incomplete proof of the specification is built in the Type Theory, whose gaps would correspond to proof obligations. On sequential imperative programs, we get the same proof obligations as those given by Floyd-Hoare logic. Compared to the latter, our approach also includes functional constructions in a straight-forward way. This work has been implemented in the Coq Proof Assistant and applied on non-trivial examples.

Introduction

The methods for proving programs developed in the last decades (see [4] for a survey), based on Floyd-Hoare logic [8, 6] or on Dijkstra's calculus of weakest preconditions [5], certainly experienced a great success. But the specification languages involved were usually low expressive first-order predicate logics which surely contributed to their relative failure in real case studies. More recent methods try to fill this gap, as for instance the B method [2], whose specification language includes a rather large part of first-order set-theory. Type Theory also provides expressive logics, well understood in theory and relatively easy to implement since they are based on a small set of rules. The Calculus of Inductive Constructions (Cic for short) is one of the most powerful logical framework of this kind, and the Coq Proof Assistant [1] is one of its implementations. Type Theory is naturally well-suited for the proof of purely functional programs. We show that it is also a good framework to specify and prove imperative programs.

There are already some formalizations of Floyd-Hoare logic in higher order logical frameworks, as for instance the work of T. Schreiber in LEGO [15] or M. Gordon in HOL [7]. Nevertheless, those formalizations still have the disadvantages of Floyd-Hoare logic: well-understood on small imperative languages, they appeared to be difficult to extend to real programming languages. For example, the few base datatypes (integers, booleans, ...) are usually not sufficient and the programmer quickly has to construct new datatypes using arrays, records,

* This research was partly supported by ESPRIT Working Group "Types".

pointers or a primitive notion of recursive datatypes. Then the extensions of Floyd-Hoare logic become painful (see [4], page 931).

For all those reasons, we think that we have to start with a more realistic language and to propose a more extensible method. We chose to consider a programming language with both functional features (functions as first-class objects, higher-order functions,...) and imperative ones (references, while loops,...). The base objects will be the ones of the Type Theory, which gives immediately a huge panel of datatypes (for instance lists, trees,... defined as inductive types in the Cic). References will be limited to purely functional values. Notice that such a language already includes FORTRAN, Pascal without an explicit use of pointers, and a rather large subset of ML.

In the traditional approach of Floyd-Hoare logic, programs are seen as state-transformers, where a state maps variables to values. Then the total correctness of a program M can be expressed in the following way:

$$\{P\}\ M\ \{Q\} \quad \equiv \quad \forall s. P(s) \Rightarrow \exists s'. \, (\llbracket M \rrbracket(s,s') \wedge Q(s,s'))$$

where $\llbracket M \rrbracket$ is the semantic interpretation of M as a relation between input and output states. It is easy to define the type of these states when they only contain integers and booleans for instance. But this becomes difficult as soon as states may contain objects of any type. Moreover, in real case studies one has to express in post-conditions that some parts of the state are not modified by the program. This is not natural.

Thinking of a variable x as an index in a global store is very close to the implementation of imperative languages, where x is a pointer in the heap. It is necessary when there is possible *aliasing* in programs i.e. when two variables may point to the same object. But in practice, most programs considered do not contain aliased variables. Then we can directly represent the *contents* of a variable x of the program by a variable of the logic. Predicates about the program's variables become predicates about the logic's variables, and not about some accessed values in a global store.

Consequently, we propose to express the semantics of an imperative program M by *a functional program* $\overline{M}$ taking as argument a tuple $\boldsymbol{x}$ of the *values* of the variables of M and returning a tuple $\boldsymbol{y}$ of the *values* of the variables (possibly) modified by M, together with the result v of the evaluation. Then, the correctness may be written

$$\{P\}\ M\ \{Q\} \quad \equiv \quad \forall \boldsymbol{x}. P(\boldsymbol{x}) \Rightarrow \exists (\boldsymbol{y}, v). \, \big((\boldsymbol{y}, v) = \overline{M}(\boldsymbol{x}) \wedge Q(\boldsymbol{x}, \boldsymbol{y}, v)\big) \qquad (1)$$

The functional translation relies on a static analysis of effects, following J.-P. Talpin and P. Jouvelot's *Type and Effect Discipline* [16], and on the use of *monads* [11, 17].

Then we propose a method to establish the correctness formula (1). To do that, we construct an incomplete proof $\widehat{M}$ of this proposition. By "incomplete" we mean that the proof term $\widehat{M}$ still contains gaps, whose types are known. Those gaps will give the proof obligations. Our work may be seen as an extension of the work of C. Parent [13] to imperative programs. Her approach to proving purely functional programs, based on realizability, consists in building an incomplete proof of the specification whose skeleton is the program itself. We extend this idea using a functional translation of imperative programs.

This paper is organized as follows. In the first section, we introduce a programming language with functional and imperative aspects, and we define a notion of *typing with effects* for this language. This allows us to define a functional translation of programs, which is proved to be semantically correct. In the next section, we introduce annotations in programs (pre- and post-conditions, variants) and we define an incomplete proof associated to each program, whose proof obligations establish the total correctness when proved. In the third section, we shortly describe the implementation of this work, which is already part of the Coq Proof Assistant [1]. At last, we will discuss related works and propose some possible extensions.

1 Effects and functional translation

Preliminary definitions and notations. In this paper we are going to consider two different kinds of programs: purely functional ones and imperative ones. Let T be the type system of purely functional programs and $\Gamma \vdash e : t$ the corresponding typing judgment, where Γ is a typing environment i.e. a mapping from variables to types. We assume that T contains at least the type of booleans and a type unit which has only one value, the constant void. The type system of imperative programs will be $T_{ref} ::= T \mid T\ \mathsf{ref}$, where $T\ \mathsf{ref}$ is the type of references over objects of type T. The corresponding typing judgment will be written $\Gamma \vdash_r e : t$. Dereferencing will be written $!x$ and assignment $x := e$, as usual in ML languages.

The abstract syntax of programs is given in Figure 1. The nonterminal symbol E stands for a purely functional expression, including possible dereferencing of variables. For the moment, we only consider call-by-value application, written $(f\ e)$ — call-by-name will be considered at the end of this section. To simplify the presentation, we chose to restrict boolean expressions appearing in if and while statements to be purely functional expressions. The precise meaning of V in the abstraction $\mathsf{fun}\ (x : V) \rightarrow M$ will be explained in the next paragraph.

1.1 Effects

The functional translation of imperative programs that we are going to define relies on a keen analysis of their *effects*. We do not intend to present new ideas or results to the existing theory of static analysis and its application to effects inference. Actually, we are only interested in determining the sets of variables possibly accessed or modified by a given program. In the general case, this would require a complex analysis of *regions*, as defined by J.-P. Talpin and P. Jouvelot in [16], or some similar technique, but since we have eliminated alias possibilities in our programs, the solution is here much simpler.

$$
\begin{array}{rcl}
M & ::= & E \\
 & | & x := M \\
 & | & M \ ; \ M \\
 & | & \mathsf{if}\ E\ \mathsf{then}\ M\ \mathsf{else}\ M \\
 & | & \mathsf{while}\ E\ \mathsf{do}\ M\ \mathsf{done} \\
 & | & \mathsf{let}\ x = \mathsf{ref}\ M\ \mathsf{in}\ M \\
 & | & \mathsf{fun}\ (x : V) \rightarrow M \\
 & | & (M\ M)
\end{array}
$$

Fig. 1. Abstract syntax of programs

The effect ϵ of a program expression will be a pair (R, W) of two sets of variables, R being the set of all the references involved in the evaluation of the expression and W the subset of R of references which are possibly modified during this evaluation. If $\epsilon_1 = (R_1, W_1)$ and $\epsilon_2 = (R_2, W_2)$ are two effects, then $\epsilon_1 \sqcup \epsilon_2$ will denote their *union* i.e. the effect $(R_1 \cup R_2, W_1 \cup W_2)$. Finally $\epsilon \backslash x$ will denote the effect obtained by removing the occurrences of x in the effect ϵ.

To do type inference with effects, we introduce a new type system composed from a type system for *values*, V, and a type system for *computations*, C. They are mutually recursively defined as follows:

$$
V \quad ::= \quad T \mid T\ \mathsf{ref} \mid V \rightarrow C \tag{2}
$$

$$
C \quad ::= \quad (V, \epsilon) \tag{3}
$$

In definition (2), we express the fact that functions are first-class values and that, since we chose call-by-value semantics, functions take values as arguments to produce computations. In definition (3), we express that a computation returns a value together with an effect. Those definitions are clearly driven by the semantic and would have been different with call-by-name semantics.

To type programs with effects, we must now consider environments mapping variables to types of values, i.e. to expressions of the type system V. If Γ is such an environment, then $\Gamma \vdash M : C$ will denote the typing judgment expressing that a program M has the type of computation C. This judgment is established by the inference rules given in Figure 2. It is clear that this judgment is decidable and that the type of computation of a program is unique.

1.2 Functional translation

The functional translation we are going to define is based on the idea of *monads*. Monads were introduced by E. Moggi [11] and P. Wadler [17], in rather different contexts. They are used by Moggi to give semantics to programming languages. The motivations of Wadler are more pragmatic and closest to ours: he uses monads to incorporate imperative aspects (stores, exceptions, input-output, ...) in purely functional languages. A monad is composed of a type operator μ and

$$\frac{\Gamma \vdash_r E : T \quad R = \{\, x \in dom(\Gamma) \mid\ !x \text{ in } E \,\}}{\Gamma \vdash E : (T, (R, \emptyset))} \ (\text{EXP})$$

$$\frac{x : T \ \mathsf{ref} \in \Gamma \quad \Gamma \vdash M : (T, (R, W))}{\Gamma \vdash x \ := \ M : (\mathsf{unit}, (\{x\} \cup R, \{x\} \cup W))} \ (\text{ASSIGN})$$

$$\frac{\Gamma \vdash M_1 : (\mathsf{unit}, \epsilon_1) \quad \Gamma \vdash M_2 : (V, \epsilon_2)}{\Gamma \vdash M_1 \ ; \ M_2 : (V, (\epsilon_1 \sqcup \epsilon_2))} \ (\text{SEQ})$$

$$\frac{\Gamma \vdash B : (\mathsf{bool}, \epsilon_0) \quad \Gamma \vdash M_1 : (V, \epsilon_1) \quad \Gamma \vdash M_2 : (V, \epsilon_2)}{\Gamma \vdash \mathsf{if}\ B\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2 : (V, (\epsilon_0 \sqcup \epsilon_1 \sqcup \epsilon_2))} \ (\text{COND})$$

$$\frac{\Gamma \vdash B : (\mathsf{bool}, \epsilon_0) \quad \Gamma \vdash M : (\mathsf{unit}, \epsilon)}{\Gamma \vdash \mathsf{while}\ B\ \mathsf{do}\ M\ \mathsf{done} : (\mathsf{unit}, (\epsilon_0 \sqcup \epsilon))} \ (\text{LOOP})$$

$$\frac{\Gamma \vdash M_1 : (T_1, \epsilon_1) \quad \Gamma, x : T_1 \ \mathsf{ref} \vdash M_2 : (T_2, \epsilon_2)}{\Gamma \vdash \mathsf{let}\ x = \mathsf{ref}\ M_1\ \mathsf{in}\ M_2 : (T_2, (\epsilon_1 \sqcup \epsilon_2) \backslash x)} \ (\text{LETREF})$$

$$\frac{\Gamma, x : V \vdash M : C}{\Gamma \vdash \mathsf{fun}\ (x : V) \rightarrow M : ((V \rightarrow C), (\emptyset, \emptyset))} \ (\text{ABS})$$

$$\frac{\Gamma \vdash M_1 : ((V \rightarrow (V_1, \epsilon_1)), \epsilon_0) \quad \Gamma \vdash M_2 : (V, \epsilon_2)}{\Gamma \vdash (M_1 \ M_2) : (V_1, (\epsilon_0 \sqcup \epsilon_1 \sqcup \epsilon_2))} \ (\text{APP})$$

Fig. 2. Typing with effects

two operators

$$\begin{aligned} &unit : A \rightarrow \mu(A) \\ &star : \mu(A) \rightarrow (A \rightarrow \mu(B)) \rightarrow \mu(B) \end{aligned}$$

satisfying three identities (which it is not necessary to give here). The main idea is that $\mu(A)$ is the type of the *computations* of type A. The *unit* operator is an injection of values into computations. The *star* operator takes a computation, evaluates it and passes its result to a function which returns a new computation. If we consider references, a possible monad is the one defined by $\mu(A) = S \rightarrow S \times A$, where S represents the store, and where *unit* and *star* are defined by

$$\begin{aligned} &unit\ v &&= \lambda s.(s, v) \\ &star\ m\ f &&= \lambda s.\mathsf{let}\ (s', v) = (m\ s)\ \mathsf{in}\ (f\ v\ s') \end{aligned}$$

In our case, such a monad is too coarse, mainly because a global store does not allow proofs of programs to be *modular* (you have to express that some parts of the store are left unchanged by the program, which is not natural and painful). So we use *local stores* i.e. tuples of values directly representing the values of *input* and *output* variables given by the inference of effects. For instance, a computation involving the references x and y of type int ref which can modify the value of x will be translated into a function taking the values of x and y as input and returning the new value of x together with the result of the computation.

As noticed by Moggi, the *star* operator really acts as the let in operator of ML. In the following we will use a let in notation to make programs more readable, but the reader should have in mind the use of the monad operator to understand the generality of the discourse.

Notations. To define the functional translation of programs, we need to manipulate collections of values and in particular to define functions taking an n-tuple representing the input of a program and returning an m-tuple representing its output. Instead of using tuples, we will rather use *records* which are easier to manipulate. The type of a record will be written $\{ x_1 : X_1; \ldots; x_n : X_n \}$ and a particular record of that type will be written $\{ x_1 = v_1; \ldots; x_n = v_n \}$, where x_i is a label, X_i its type and v_i a value of that type. Records will be written $\boldsymbol{x}, \boldsymbol{y}, \ldots$ for convenience and $\boldsymbol{x}.l$ will denote the field of label l in record $\boldsymbol{x}$. We define the operation $\oplus$ on records as follows: the record $\boldsymbol{x} \oplus \boldsymbol{y} = \{ z_1 = w_1; \ldots; z_k = w_k \}$ contains all the labels of $\boldsymbol{x}$ and $\boldsymbol{y}$, and w_i is equal to $\boldsymbol{y}.z_i$ if z_i is a label of $\boldsymbol{y}$ and to $\boldsymbol{x}.z_i$ otherwise. In other words, it is the update of the record $\boldsymbol{x}$ by the record $\boldsymbol{y}$ i.e. the record made with the fields of $\boldsymbol{y}$ when they exist and of $\boldsymbol{x}$ in the other case. At last, $\boldsymbol{x} \backslash l$ will denote the record $\boldsymbol{x}$ in which the field l is removed and $\boldsymbol{x}[l \leftarrow l']$ will denote the record $\boldsymbol{x}$ where the label l is renamed into l'.

First, we give the interpretation of types with effects as purely functional types.

Definition 1. *The functional translation of types of values and types of computations are mutually recursively defined as follows:*

$$
\begin{array}{lll}
\textit{values} & \overline{T} & = \; T \\
 & \overline{V \to C} & = \; \overline{V} \to \overline{C} \\
\\
\textit{computations} & \overline{(V, (R, W))} & = \; \overline{R} \to \overline{W} \times \overline{V} \\
\\
\textit{effects} & \overline{\{x_1, \ldots, x_n\}} & = \; \{x_1 : T_1; \ldots; x_n : T_n\} \\
 & & \quad\quad \textit{where } x_i \textit{ has type } T_i \ \mathsf{ref}
\end{array}
$$

If Γ is a typing environment mapping the x_i's to the types V_i, then $\overline{\Gamma}$ will denote the environment mapping the x_i's to the types $\overline{V_i}$.

Note, there is no translation for the types T ref: indeed, there is no program expression of such type and therefore there is no counterpart in the functional world. References, seen as *pointers*, do not exist anymore after the translation since we only manipulate the *values* they contain.

We are now in the position to define the functional translation itself. In the following we will use a slight abuse of notations: we will write $(\overline{M}\ \boldsymbol{x})$ even when the record $\boldsymbol{x}$ should be restricted to a subset of its fields (Another possibility is to consider that we have sub-typing on records).

Definition 2. *Let M be a program of type C in a context Γ. Then the functional translation of M, written $\overline{M}$, is a functional program of type $\overline{C}$ in the context $\overline{\Gamma}$, which is defined by induction on the structure of M as follows:*

- $\underline{M \equiv E}$:

$$\overline{M} = \lambda \boldsymbol{x}.(\{\}, E[!x \leftarrow \boldsymbol{x}.x])$$

- $\underline{M \equiv x_0 \ := \ M_1}$:

$$\overline{M} = \lambda \boldsymbol{x}.\mathsf{let}\ (\boldsymbol{x_1}, v) = (\overline{M_1}\ \boldsymbol{x})\ \mathsf{in}\ (\boldsymbol{x_1} \oplus \{\, x_0 = v \,\}, \mathsf{void})$$

- $\underline{M \equiv M_1\ ;\ M_2}$:

$$\begin{aligned}\overline{M} = \lambda \boldsymbol{x}.\ & \mathsf{let}\ (\boldsymbol{x_1}, \_) = (\overline{M_1}\ \boldsymbol{x})\ \mathsf{in} \\ & \mathsf{let}\ (\boldsymbol{x_2}, v) = (\overline{M_2}\ (\boldsymbol{x} \oplus \boldsymbol{x_1}))\ \mathsf{in} \\ & (\boldsymbol{x_1} \oplus \boldsymbol{x_2}, v)\end{aligned}$$

- $\underline{M \equiv \mathsf{if}\ E\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2}$:

$$\begin{aligned}\overline{M} = \lambda \boldsymbol{x}.\ & \mathsf{if}\ (\overline{E}\ \boldsymbol{x})\ \mathsf{then} \\ & \quad \mathsf{let}\ (\boldsymbol{x_1}, v) = (\overline{M_1}\ \boldsymbol{x})\ \mathsf{in}\ (\boldsymbol{x} \oplus \boldsymbol{x_1}, v) \\ & \mathsf{else} \\ & \quad \mathsf{let}\ (\boldsymbol{x_1}, v) = (\overline{M_2}\ \boldsymbol{x})\ \mathsf{in}\ (\boldsymbol{x} \oplus \boldsymbol{x_1}, v)\end{aligned}$$

  It is clear now why we impose the condition on output variables to be included in the input variables: indeed, in both branches of the if we must return values for the same set of variables. So, when a variable is not modified by a branch, we have to return its initial value: therefore it must belong to the input values.

- $\underline{M \equiv \mathsf{while}\ E\ \mathsf{do}\ M_1\ \mathsf{done}}$:

  To translate a loop, we use the following semantic equivalence:

$$M \approx (Y\ (\lambda w : \mathsf{unit} \rightarrow \mathsf{unit}.\mathsf{if}\ E\ \mathsf{then}\ (M_1\ ;\ (w\ \mathsf{void}))\ \mathsf{else}\ \mathsf{void})\ \mathsf{void})$$

  where Y is a fixpoint operator. Then

$$\begin{aligned}\overline{M} = (Y\ \lambda w : \overline{C}.\lambda \boldsymbol{x}.\ & \mathsf{if}\ (\overline{E}\ \boldsymbol{x})\ \mathsf{then} \\ & \quad \mathsf{let}\ (\boldsymbol{x_1}, \_) = (\overline{M_1}\ \boldsymbol{x})\ \mathsf{in}\ \ (w\ (\boldsymbol{x} \oplus \boldsymbol{x_1})) \\ & \mathsf{else} \\ & \quad (\boldsymbol{x}, \mathsf{void}))\end{aligned}$$

  Here we assume that we have a fixpoint operator Y in the Calculus of Constructions, which is not the case usually. This need will disappear when we are in position to establish termination, as it is the case in the next section.

- $\underline{M \equiv \mathsf{let}\ x_0 = \mathsf{ref}\ M_1\ \mathsf{in}\ M_2}$:

$$\begin{aligned}\overline{M} = \lambda \boldsymbol{x}.\ & \mathsf{let}\ (\boldsymbol{x_1}, v_1) = (\overline{M_1}\ \boldsymbol{x})\ \mathsf{in} \\ & \mathsf{let}\ (\boldsymbol{x_2}, v_2) = (\overline{M_2}\ (\boldsymbol{x} \oplus \boldsymbol{x_1} \oplus \{\, x_0 = v_1 \,\}))\ \mathsf{in} \\ & (\boldsymbol{x_1} \oplus (\boldsymbol{x_2} \backslash x_0), v_2)\end{aligned}$$

- $\underline{M \equiv \mathsf{fun}\ (x : V) \rightarrow M_1}$:

$$\overline{M} = \lambda x : \overline{V}.\overline{M_1}$$

– $\underline{M \equiv (M_1\ M_2)}$

$$\begin{aligned}\overline{M} = \lambda \boldsymbol{x}.\ & \mathsf{let}\ (\boldsymbol{x_1}, a) = (\overline{M_2}\ \boldsymbol{x})\ \mathsf{in} \\ & \mathsf{let}\ (\boldsymbol{x_2}, f) = (\overline{M_1}\ (\boldsymbol{x} \oplus \boldsymbol{x_1}))\ \mathsf{in} \\ & \mathsf{let}\ (\boldsymbol{x_3}, v) = (f\ a\ (\boldsymbol{x} \oplus \boldsymbol{x_1} \oplus \boldsymbol{x_2}))\ \mathsf{in} \\ & (\boldsymbol{x_1} \oplus \boldsymbol{x_2} \oplus \boldsymbol{x_3}, v)\end{aligned}$$

Notice that we chose a semantics where the function is evaluated *after* its argument; therefore multiple arguments are evaluated from right to left.

□

To justify this definition, we have to prove that $\overline{M}$ preserves the semantics of M. We chose the formal semantics introduced by A. K. Wright and M. Felleisen in [18] to prove type soundness of SML with references and exceptions. The idea is to introduce a syntactic distinction between values and expressions and to extend the syntax of programs with a new construction $\rho\theta.M$, where θ is a mapping from variables to values representing the *store*. Then small-step reductions are introduced on extended programs, as rewriting rules driven by the syntax, and the evaluation relation $M \leadsto \rho\theta'.v$ is defined as the transitive closure of those reductions.

The preservation of the semantics can be expressed by the following theorem:

Theorem 1. *Let Γ be a well-formed environment whose references are $x_1 : T_1\ \mathsf{ref}, \ldots, x_n : T_n\ \mathsf{ref}$ and M a program such that $\Gamma \vdash M : (V, (R, W))$. Let v_i be values of types T_i and θ the store mapping the x_i's to the v_i's. Then*

$$\rho\theta.M \leadsto \rho\theta'.v \qquad \Longleftrightarrow \qquad \overline{M}(\overline{\theta(R)}) = (\overline{\theta'(W)}, \overline{v})$$

Proof. We won't enter here the details of this proof. The *if* part is proved by induction on the derivation of the evaluation relation and the *only if* part is proved by induction on the program M. In both cases the proof is rather systematic since the semantic relation is driven by the syntax of programs. The reader may consult [18] to get the formal semantics of ML with references, which is greatly simplified in our case since we do not have polymorphism. □

Call-by-variable

Until now, we did not consider call-by-variable arguments i.e. the passing of references to functions, because their treatment is rather complex when combined with partial application (it requires the use of painful explicit coercions to re-organize elements in tuples). But if we do not allow partial application of functions doing side-effects, we can simply deal with call-by-variable. To give an idea of this method, let us consider the simple case of a function M taking a first argument by-value and a second one by-variable i.e. of type $V_1 \to (x : T_2\ \mathsf{ref}) \to (T, \epsilon)$. Its second argument is given a name, x, since it may appear in ϵ. Then the typing rule for the application $(M\ M_1\ z)$ is the following

$$\frac{\Gamma \vdash M : ((V_1 \to (x : T_2\ \mathsf{ref}) \to (T, \epsilon)), \epsilon_0) \quad \Gamma \vdash M_1 : (V_1, \epsilon_1) \quad z : T_2\ \mathsf{ref} \in \Gamma}{\Gamma \vdash (M\ M_1\ z) : T, (\epsilon_0 \sqcup \epsilon_1 \sqcup \epsilon[x \leftarrow z])}$$

During the functional translation, the second argument of M disappears: indeed, the functional program $\overline{M}$ does not manipulate references, but only their values, and consequently $\overline{M}$ has type $\overline{V_1} \rightarrow \overline{R} \rightarrow \overline{W} \times T$, where $\epsilon = (R, W)$. The reference x now appears as a field of $\overline{R}$ and $\overline{W}$. So the functional translation of the application $(M\ M_1\ z)$ will be

$$
\begin{array}{l}
\lambda \boldsymbol{x}.\,\mathsf{let}\ (\boldsymbol{x_1}, v_1) = (\overline{M_1}\ \boldsymbol{x})\ \mathsf{in} \\
\quad \mathsf{let}\ (\boldsymbol{x_2}, f) = (\overline{M}\ (\boldsymbol{x} \oplus \boldsymbol{x_1}))\ \mathsf{in} \\
\quad \mathsf{let}\ (\boldsymbol{x_3}, v) = (f\ v_1\ (\boldsymbol{x} \oplus \boldsymbol{x_1} \oplus \boldsymbol{x_2})[z \leftarrow x])\ \mathsf{in} \\
\quad (\boldsymbol{x_1} \oplus \boldsymbol{x_2} \oplus \boldsymbol{x_3}[x \leftarrow z], v)
\end{array}
$$

This example is easily generalized to arbitrary numbers of call-by-value and call-by-variable arguments.

2 Program Correctness

We now come to the main point, proofs of programs, and the first thing to do is to define *how to specify* the programs. In almost all the literature about Floyd-Hoare logic and related systems, the programs are purely imperative and therefore can be seen as *sequences of statements* with separated notions of statements and expressions. Typically we have an abstract syntax of the kind

$$S \quad ::= \quad \mathsf{skip} \mid x := E \mid S\ ;\ S \mid \mathsf{if}\ E\ \mathsf{then}\ S\ \mathsf{else}\ S \mid \mathsf{while}\ E\ \mathsf{do}\ S\ \mathsf{done}$$

where E is a pre-defined notion of expressions. Then it is natural to specify them by inserting logical assertions *between* the successive evaluations i.e. between the statements.

But in our case, following the tradition of functional programming languages, the notions of programs and expressions are identified. For instance, we can write programs like

$$x := (x := !x + 1\ ;\ !x)$$

$$x := (\mathsf{if}\ B\ \mathsf{then}\ \ldots\ \mathsf{else}\ \ldots)$$

This may appear superfluous but we claim that this is the key to deal with functions without difficulty. Take for instance a statement like

$$x := (f\ a)$$

where f is a function which possibly has side-effects. Since our abstract syntax already includes statements of the form $x := M$ where M is an arbitrary program, the above assignment does not change the effects. In other words, inlining the function f would give a correct program, which is not the case in traditional frameworks.

Since the notions of programs and expressions are identified, we propose a more general way to specify programs, where each sub-expression of a program may be annotated with a pre- and a post-condition. A pre-condition will be a predicate over the current values of variables, as usual. For the post-conditions, we will use before-after predicates i.e. predicates referring to the current values of variables and to their values before the evaluation of the expression, as it is done in VDM. The current value of a variable x will be written x in both pre- and post-conditions and its value before the evaluation will be written $\overleftarrow{x}$ in a post-condition. Moreover, a post-condition must be able to express properties of the *result* of the evaluation, which will be given a name in post-conditions. The abstract syntax of programs is extended as follows:

$$\begin{array}{lll} M & ::= & \{P\}\ S\ \{v \mid Q\} \\ S & ::= & E \mid x := M \mid M \ ;\ M \mid \ldots \text{(as formerly)} \end{array}$$

where $\{P\}$ stands for a pre-condition and $\{x \mid Q\}$ for a post-condition in which the result is referred as x. The type systems V and C for values and computations are enriched consequently:

$$V \quad ::= \quad T \mid T \ \mathsf{ref} \mid (x : V) \to C \tag{4}$$

$$C \quad ::= \quad (v : V, \epsilon, P, Q) \tag{5}$$

Note that function arguments are given names since they may now appear in annotations.

As previously, we have to give first an interpretation of types in the target language, the Calculus of Inductive Constructions. Those interpretations are written $\widehat{V}$ and $\widehat{C}$, $\widehat{C}$ being the *correctness formula.* A first idea for $\widehat{C}$ could be

$$\forall \boldsymbol{x}.\, P(\boldsymbol{x}) \Rightarrow \exists(\boldsymbol{y}, v).\, (\boldsymbol{y}, v) = \overline{M}(\boldsymbol{x}) \wedge Q(\boldsymbol{x}, \boldsymbol{y}, v) \tag{6}$$

But $\overline{M}$ is usually not a total function and therefore is not definable in the Cic. Thus, we choose to define $\widehat{C}$ just as

$$\forall \boldsymbol{x}.\, P(\boldsymbol{x}) \Rightarrow \exists(\boldsymbol{y}, v).\, Q(\boldsymbol{x}, \boldsymbol{y}, v) \tag{7}$$

and we will construct a particular proof of (7), $\widehat{M}$, which has the property to have a computational contents equal to $\overline{M}$. Therefore, the realizability theorem [14] will exactly express (6), which is the expected result.

Definition 3. *The interpretation in* Cic *of the types of values and the types of computations are defined as follows:*

$$\begin{array}{llll} \textit{if} & V = T & \textit{then} & \widehat{V} = T \\ \textit{if} & V = (x : V_0) \to C & \textit{then} & \widehat{V} = \forall x : \widehat{V_0}.\, \widehat{C} \end{array}$$

$$\begin{array}{ll} \textit{if} & C = (r : V, (R, W), P, Q) \\ \textit{then} & \widehat{C} = \forall \boldsymbol{x} : \overline{R}.\, P(\boldsymbol{x}) \Rightarrow \exists(\boldsymbol{y}, v) : \overline{W} \times \widehat{V}.\, Q(\boldsymbol{x}, \boldsymbol{y}, v) \end{array}$$

If Γ is a typing environment mapping the x_i's to the types V_i, then $\widehat{\Gamma}$ will denote the environment mapping the x_i's to the types $\widehat{V_i}$.

Before giving the translation of the programs themselves, we have to solve a last problem: since functions in the CIC are necessarily total, a loop cannot be simply translated using a general fixpoint operator. Moreover, we are interested in proving *total correctness* and therefore we have to justify the termination of loops. Thus, loops will be annotated with a well-foundedness argument:

$$\mathsf{while}_{\phi,R}\ E\ \mathsf{do}\ M\ \mathsf{done}$$

where ϕ is a term and R a relation over the type of ϕ. Then such a loop will be translated using a well-founded induction over ϕ.

In the following, an incomplete proof term of the CIC is a term where some sub-terms are still undefined and written "?". We assume that those gaps are typed, but we will not write the corresponding types for a greater clarity.

Definition 4. *Let M be a program of type C in a context Γ. Then the interpretation of a program M in the CIC, written $\widehat{M}$, is an incomplete proof term of type $\widehat{C}$ in the context $\widehat{\Gamma}$, which is defined by induction on the structure of M as follows:*

– $\underline{M \equiv \{P\}\ E\ \{Q\}}$:

$$\widehat{M} = \lambda \boldsymbol{x}.\lambda h : P.(\{\}, E[!x \leftarrow \boldsymbol{x}.x], ?)$$

– $\underline{M \equiv \{P\}\ x_0\ := \ M_1\ \{Q\}}$:

$$\widehat{M} = \lambda \boldsymbol{x}.\lambda h : P.\mathsf{let}\ (\boldsymbol{x_1}, v, q_1) = (\widehat{M_1}\ \boldsymbol{x}\ ?)\ \mathsf{in}\ (\boldsymbol{x_1} \oplus \{\, x_0 = v\,\}, \mathsf{void}, ?)$$

– $\underline{M \equiv \{P\}\ M_1\ ;\ M_2\ \{Q\}}$:

$$\begin{aligned}\widehat{M} = \lambda \boldsymbol{x}.\lambda h : P.\ &\mathsf{let}\ (\boldsymbol{x_1}, \_, q_1) = (\widehat{M_1}\ \boldsymbol{x}\ ?)\ \mathsf{in}\\ &\mathsf{let}\ (\boldsymbol{x_2}, v, q_2) = (\widehat{M_2}\ (\boldsymbol{x} \oplus \boldsymbol{x_1})\ ?)\ \mathsf{in}\\ &(\boldsymbol{x_1} \oplus \boldsymbol{x_2}, v, ?)\end{aligned}$$

– $\underline{M \equiv \{P\}\ \mathsf{if}\ E\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2\ \{Q\}}$:

$$\begin{aligned}\widehat{M} = \lambda \boldsymbol{x}.\lambda h : P.\ &\mathsf{if}\ (\widehat{E}\ \boldsymbol{x})\ \mathsf{then}\\ &\quad \mathsf{let}\ (\boldsymbol{x_1}, v, q) = (\widehat{M_1}\ \boldsymbol{x}\ ?)\ \mathsf{in}\ (\boldsymbol{x} \oplus \boldsymbol{x_1}, v, ?)\\ &\mathsf{else}\\ &\quad \mathsf{let}\ (\boldsymbol{x_1}, v, q) = (\widehat{M_2}\ \boldsymbol{x}\ ?)\ \mathsf{in}\ (\boldsymbol{x} \oplus \boldsymbol{x_1}, v, ?)\end{aligned}$$

– $\underline{M \equiv \{P\}\ \mathsf{while}_{\phi,R}\ E\ \mathsf{do}\ M_1\ \mathsf{done}\ \{Q\}}$:
To construct the proof term corresponding to this loop, we are going to use a well-founded induction over ϕ. The well-founded recursor $Y_{R,\pi}$ is a higher-order term taking a relation R and a proof π that R is well-founded. In our case, the proof π is replaced by a proof obligation.

$$\begin{aligned}\widehat{M} = \lambda \boldsymbol{x}.((Y_{R,?}\ &\lambda w.\lambda \boldsymbol{x}.\lambda \psi.\lambda h : P.\\ &\mathsf{if}\ (\widehat{E}\ \boldsymbol{x})\ \mathsf{then}\\ &\quad \mathsf{let}\ (\boldsymbol{x_1}, \_, q) = (\widehat{M_1}\ \boldsymbol{x}\ ?)\ \mathsf{in}\ \ (w\ (\boldsymbol{x} \oplus \boldsymbol{x_1})\ \phi(\boldsymbol{x} \oplus \boldsymbol{x_1})\ ?\ ?)\\ &\mathsf{else}\\ &\quad (\boldsymbol{x}, \mathsf{void}, ?))\ \boldsymbol{x}\ \phi(\boldsymbol{x}))\end{aligned}$$

Here we have used P as the loop *invariant*. There are four proof obligations related to that loop: to prove that R is well-founded; to prove that ϕ strictly decreases in M_1; to prove that M_1 preserves P; and to prove that P implies Q when the test is negative.

– $\underline{M \equiv \{P\} \ \mathsf{let}\ x_0 = \mathsf{ref}\ M_1 \ \mathsf{in}\ M_2 \ \{Q\}}$:

$$\begin{aligned}\widehat{M} = \lambda \boldsymbol{x}.\lambda h : P.\, & \mathsf{let}\ (\boldsymbol{x_1}, v_1, q_1) = (\widehat{M_1}\ \boldsymbol{x}\ ?)\ \mathsf{in} \\ & \mathsf{let}\ (\boldsymbol{x_2}, v_2, q_2) = (\widehat{M_2}\ (\boldsymbol{x} \oplus \boldsymbol{x_1} \oplus \{\, x_0 = v_1 \,\})\ ?)\ \mathsf{in} \\ & (\boldsymbol{x_1} \oplus (\boldsymbol{x_2} \backslash x_0), v_2, ?)\end{aligned}$$

– $\underline{M \equiv \{P\} \ \mathsf{fun}\ (x : V) \rightarrow M_1 \ \{Q\}}$:

$$\widehat{M} = \lambda h : P.((\lambda x : \widehat{V}.\widehat{M_1}), ?)$$

Notice that here P and Q are the pre- and post-conditions of the *function* and not of the result of its application. Usually they are empty.

– $\underline{M \equiv \{P\} \ (M_1 \ M_2) \ \{Q\}}$

$$\begin{aligned}\widehat{M} = \lambda \boldsymbol{x}.\lambda h : P.\, & \mathsf{let}\ (\boldsymbol{x_1}, a, q_1) = (\widehat{M_2}\ \boldsymbol{x}\ ?)\ \mathsf{in} \\ & \mathsf{let}\ (\boldsymbol{x_2}, f, q_2) = (\widehat{M_1}\ (\boldsymbol{x} \oplus \boldsymbol{x_1})\ ?)\ \mathsf{in} \\ & \mathsf{let}\ (\boldsymbol{x_3}, v, q_3) = (f\ a\ (\boldsymbol{x} \oplus \boldsymbol{x_1} \oplus \boldsymbol{x_2})\ ?)\ \mathsf{in} \\ & (\boldsymbol{x_1} \oplus \boldsymbol{x_2} \oplus \boldsymbol{x_3}, v, ?)\end{aligned}$$

□

To justify the validity of the proof obligations we obtain, we have to prove that they apply to the right values. It directly results from Theorem 1 and from the following proposition (which is immediate by construction of $\widehat{M}$):

Proposition 1. *For any program M the following equality holds*

$$\mathcal{E}(\widehat{M}) = \overline{M} \tag{8}$$

where $\mathcal{E}$ is the extraction *operator, which computes the informative contents of a proof (extraction in the Calculus of Inductive Constructions was introduced by C. Paulin in [14]).*

Examples and comparison to Floyd-Hoare logic

Example 1. Let us consider the classical case of assignment where the right hand side has no effect and no annotation i.e.

$$M \quad \equiv \quad \{P\}\ x_0 := E\ \{Q\}$$

Then the proof term $\widehat{M}$ reduces to $\lambda \boldsymbol{x}.\lambda h : P(\boldsymbol{x}).(\boldsymbol{x} \oplus \{\, x_0 = E \,\}, \mathsf{void}, ?)$ where the gap has type $Q(\boldsymbol{x}, \boldsymbol{x} \oplus \{\, x_0 = E \,\}, \mathsf{void})$. So we get only one proof obligation, which is the following, with some abuses of notation:

$$P \Rightarrow Q[x_0 \leftarrow E]$$

This is exactly the one we get in Floyd-Hoare logic (with the combination of the consequence rule and the assignment rule). □

Example 2. Let us consider again an assignment, but where the expression assigned is now the result of a function call, that is $M \equiv \{P\}\ x_0 := (f\ E)\ \{Q\}$. The function f has a type V of the form $V_0 \to (v : V_1, \epsilon, P_f, Q_f)$. Therefore, we can see the program M as annotated this way:

$$M \quad \equiv \quad \{P\}\ x_0 := \{P_f\}\ (f\ E)\ \{v \mid Q_f\}\ \{Q\}$$

Then if we look at the proof term $\widehat{M}$, we find two gaps, the first one corresponding to the pre-condition P_f and the second one to the post-condition Q. More precisely, the two proof obligations are the following:

$$P \Rightarrow P_f \qquad \text{and} \qquad P \Rightarrow P_f \Rightarrow Q_f \Rightarrow Q[x_0 \leftarrow v]$$

Contrary to the previous case, the expression assigned is no more substituted in the post-condition but *abstracted* through the variable v. □

3 Implementation

This work has been implemented in the Coq Proof Assistant [1] and is currently released with the system, together with a documentation and a few examples. The user gives an annotated imperative program to a tactic, and a set of proof obligations are produced, which must be proved to complete the correctness proof. As explained at the end of the first section, we have call-by-variable but without partial application (so it is closer to Pascal than ML regarding this point). The implementation also includes some additional features, namely arrays and a way to eradicate the use of auxiliary variables.

Arrays. When dealing with arrays, we face a potential source of aliasing problems, since $t[i]$ and $t[j]$ may refer to the same cell of an array, while it is not possible to decide statically if i is equal to j or not. A standard solution is to consider *the whole array* as a mutable data, like any reference (see [4], page 931). Consequently, programs manipulating arrays become functions taking arrays as arguments and returning new arrays. Arrays are axiomatized in the Cic in the very simplest way, and proof obligations are produced to check that indexes are always within the bounds of arrays.

The case of auxiliary variables. Auxiliary variables, sometimes called *logical variables*, are used in specifications to relate values of variables at different moments of the execution of a program. Indeed, the before-after predicates used in post-conditions are not always powerful enough to express the specifications (see for instance [4], page 940). Although it is possible to give a formal interpretation to auxiliary variables, as shown by T. Schreiber in [15], we propose another way to solve this problem. The idea is very simple: since our functional translation define names for each new value of a variable (at each application of the monad let in operator), we just have to allow the user to access those names. So we added the possibility to put labels inside programs (like the ones used for the goto statement in old programming languages), and the user may refer to the value of a given variable at a particular moment using the associated label. Then the substitution inside pre- and post-conditions is performed during the construction of $\widehat{M}$. This is a rather technical point, but it appeared to be very useful in practice, and the proof obligations generated are simpler to prove since they do not require rewriting.

Case studies. The first case study we did with our implementation is the program *find*, a quite complex algorithm which was proved correct by Hoare using his axiomatic logic in 1971 [9]. Applied to that annotated program, our method generated *exactly the same proof obligations* than the ones given in Hoare's paper. We also did a correctness proof of the *quicksort* algorithm and of the Knuth-Pratt-Morris string searching algorithm. Another case study is a proof of *insertion sort*, which was done recently by a novel user with the Coq system. Those case studies cover all the functionalities of the method, including loops, procedures and recursive functions.

4 Discussion and future work

We have set out in this paper a method to establish the total correctness of imperative programs in Type Theory. It shares the goals of traditional Floyd-Hoare logic, regarding programs annotations and proof obligations we expect to get, but this method uses a completely different approach, based on the construction of an incomplete proof of its correctness, itself built on top of a functional translation of the imperative program. On sequential imperative programs, we get in practice the same proof obligations as Floyd-Hoare logic, which was the expected result.

Compared to the B method of J.-R. Abrial [2], our method, even if it has not the same level of maturity, offers some benefits. First, it is based on Type Theory, a framework which is small, powerful and well-understood on a theoretical point-of-view. Secondly, we are able to express the correctness of a program and its proof in the logic, which is not the case in traditional methods where programs, their specifications and their proofs live in different worlds and where correctness depends on the correctness of the tools relating those parts. The B method was recently partly formalized in Isabelle/HOL by P. Chartier [3]. But such a formalization has the same drawbacks as Floyd-Hoare logic formalizations already mentioned [7, 15]: they may be useful to prove theoretical properties of the framework but can not be applied in practice because of painful encodings.

Closest to our approach is the one of C. Muñoz [12], who formalized the B abstract machines in PVS. As we do, he directly manipulates objects of the Type Theory, which extends the range of datatypes of B, and uses record dependent types to define the states of abstract machines (a collection of variables with a global invariant). Then operations are seen as functions on states, and the preservation of the invariant is expressed by typing, proof obligations being generated by the *type-checking conditions* mechanism of PVS.

Regarding future works, we would like to incorporate pattern-matching and exceptions to reach a more realistic language. While pattern-matching is only technical — semantically, it can be viewed as a combination of tests and access functions — the treatment of exceptions needs an extension of the notion of effects and of the specification. However, we think that our framework is well-suited for such an extension, since the only thing to do is to define the corresponding monad.

References

1. The Coq Proof Assistant. `http://coq.inria.fr/`.
2. J. R. Abrial. *The B-Book. Assigning programs to meaning.* Cambridge University Press, 1996.
3. P. Chartier. Formalization of B in Isabelle/HOL. In *Proceedings of the Second B International Conference*, Montpellier, France, April 1998. Springer Verlag LNCS 1393.
4. P. Cousot. *Handbook of Theoretical Computer Science*, volume B, chapter Methods and Logics for Proving Programs, pages 841–993. Elsevier Science Publishers B. V., 1990.
5. E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.
6. R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science, Proceedings of Symposia in applied Mathematics 19*, pages 19–32, Providence, 1967. American Mathematical Society.
7. Mike Gordon. *Teaching and Learning Formal Methods*, chapter Teaching hardware and software verification in a uniform framework. Academic Press, 1996.
8. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, 1969. Also in [10] pages 45–58.
9. C. A. R. Hoare. Proof of a program : *Find. Communications of the ACM*, 14(1):39–45, January 1971. Also in [10] pages 59–74.
10. C. A. R. Hoare and C. B. Jones. *Essays in Computing Science.* Prentice Hall, New York, 1989.
11. E. Moggi. Computational lambda-calculus and monads. In *IEEE Symposium on Logic in Computer Science*, 1989.
12. C. Muñoz. Supporting the B-method in PVS: An Approach to the Abstract Machine Notation in Type Theory. Submitted to FSE-98, 1998.
13. C. Parent. Developing certified programs in the system Coq – The Program tactic. Technical Report 93-29, Ecole Normale Supérieure de Lyon, October 1993. Also in Proceedings of the BRA Workshop Types for Proofs and Programs, May 93.
14. C. Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, Austin, January 1989. ACM.
15. T. Schreiber. Auxiliary Variables and Recursive Procedures. In *TAPSOFT'97: Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 697–711. Springer-Verlag, April 1997.
16. J.-P. Talpin and P. Jouvelot. The Type and Effect Discipline. *Information and Computation*, 111(2):245–296, 1994.
17. P. Wadler. Monads for functional programming. In *Proceedings of the Marktoberdorf Summer School on Program Design Calculi*, August 1992.
18. A. K. Wright and M. Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, 115:38–94, 1994.