Proof of Imperative Programs in Type Theory.

Jean-Christophe Filliatre

Laboratoire de l'Informatique du Parallélisme

Ecole Normale Supérieure de Lyon
Unité de recherche associée au CNRS n°1398

Proof of Imperative Programs in Type Theory

Jean-Christophe Filliâtre

July 1997

Research Report N$^{\text{o}}$ 97-24

Ecole Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : (+33) (0)4.72.72.80.00 Télécopieur : (+33) (0)4.72.72.80.80
Adresse électronique : lip@lip.ens−lyon.fr

Proof of Imperative Programs in Type Theory

Jean-Christophe Filliâtre

July 1997

Abstract

Proofs of correctness of imperative programs are traditionally done in first order frameworks derived from Hoare logic [8]. On the other hand, correctness proofs of purely functional programs are almost always done in higher order logics. In particular, the realizability [10] allow to extract correct functional programs from constructive proofs of existential formulae. In this paper, we establish a relation between these two approaches and show how proofs in Hoare logic can be interpreted in type theory, yielding a translation of imperative programs into functional ones. Starting from this idea, we propose an interpretation of correctness formulae in type theory for a programming language mixing imperative and functional features. One consequence is a good and natural solution to the problems of procedures and side-effects in expressions.

Keywords: Program validation, Hoare logic, Realizability, Type Theory

Résumé

Les preuves de correction de programmes impératifs sont traditionnellement faites dans des théories du premier ordre dérivées de la logique de Hoare [8]. D'un autre côté, les preuves de correction de programmes purement fonctionnels sont le plus souvent faites dans des formalismes d'ordre supérieur. En particulier, la réalisabilité [10] permet d'extraire des programmes fonctionnels corrects à partir de preuves constructives de formules existentielles. Dans ce papier, nous établissons une relation entre ces deux approches et montrons comment les preuves en logique de Hoare peuvent être interprétées en théories des types, conduisant à une traduction fonctionnelle des programmes impératifs. Partant de cette idée, nous proposons une interprétation des formules de correction en théorie des types pour un langage de programmation mélangeant des traits impératifs et fonctionnels. Une conséquence de cette interprétation est une solution simple et naturelle aux problèmes des procédures et des effets de bord dans les expressions.

Mots-clés: Validation de programmes, Logique de Hoare, Réalisabilité, Théorie des Types

Proof of Imperative Programs in Type Theory

Jean-Christophe Filliâtre*

LIP, URA CNRS 1398, ENS Lyon

46 Allée d'Italie, 69364 Lyon cedex 07, France

e-mail: Jean-Christophe.Filliatre@ens-lyon.fr

September 5, 1997

Abstract

Proofs of correctness of imperative programs are traditionally done in first order frameworks derived from Hoare logic [8]. On the other hand, correctness proofs of purely functional programs are almost always done in higher order logics and are based on the notion of realizability [10]. In this paper, we establish a relation between these two approaches and show how proofs in Hoare logic can be interpreted in type theory, yielding a translation of imperative programs into functional ones. Starting from this idea, we propose an interpretation of correctness formulae in type theory for a programming language mixing imperative and functional features. One consequence is a good and natural solution to the problems of procedures and side-effects in expressions.

Introduction

After having remained unexploited for a long time, the formalism known as Hoare logic has finally ended up in formal specification languages, like Z [7] or VDM [9], and more recently in real implementations of formal software validation methods, like the KIV project [15] or the B method [1]. The programming languages handled by such methods are imperative and the underlying logic appears to be mainly a first-order predicate calculus, usually based on a set theoretical framework.

Type theory is rather used to deal with correctness proof of purely functional programs, because of the deep relation between typing and natural deduction — the so-called Curry-Howard isomorphism (see [6] for a good introduction to type theory). Moreover, the computational content of proofs in type theory, expressed by the notion of realizability, is naturally written as a functional program.

Actually, we can establish some connections between traditional Hoare logic and the notion of realizability. This relation naturally introduces a functional translation of imperative programs, which is not like the one given by a traditional denotational semantics, but which

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

yields programs rather close to the one we would have written ourselves. Based on those ideas, we propose a new interpretation of the correctness formula in type theory, with corresponding deduction rules "à la Hoare". With such an interpretation, it is easy to extend the programming language with functional features, and in particular with procedures and functions, which had never been easily handled by traditional Hoare logic.

This article is organized as follows. In the first section, we quickly describe the two main approaches to program validation i.e. Hoare logic on one hand and realizability on the other hand. Then we show how they actually relate and what we can learn out of this relation. In the second section, we propose an interpretation of the correctness formula in type theory and we give a corresponding set of deduction rules which is correct and complete. In the third section, we extend the programming language with functional features, giving a first way to reason about procedures and functions. In the fourth section, we give a better way to handle procedures and functions in structured programs and this allow the treatment of recursive functions. Finally, we compare our approach of software validation to the traditional ones and we discuss about the remaining work to get a real environment for program validation based on the Calculus of Inductive Constructions as specification language and on the Coq Proof Assistant as prover.

1 Hoare Logic and Realizability

In this section, we shall compare the two traditional approaches of program validation for both imperative and functional programs. The relation that comes out of this comparison will be the starting point of a new proposition for expressing the correctness of programs.

1.1 Imperative programs and Hoare logic.

In the traditional approach to showing the total correctness of imperative program, the formal semantics of a program p is defined as a relation between p and two stores σ and τ, which states that the evaluation of p on the store σ will terminate, with the resulting store as τ. Let us denote this relationship $< p, \sigma > \rightarrow \tau$. There are many ways to define this relation, depending on the programming language. Here, for simplicity of presentation, let us consider the imperative language with syntax:

$$C \quad ::= \quad \mathsf{skip} \mid x := I \mid C \ ; \ C \mid \mathsf{if}\ B\ \mathsf{then}\ C\ \mathsf{else}\ C \mid \mathsf{while}_\phi\ B\ \mathsf{do}\ C\ \mathsf{done}$$

where I stands for an integer expression and B for a boolean expression. Since we are interested in total correctness of programs, the while construct is annotated by a measure ϕ, which can be supposed here to be a natural number. The usual semantics of such a programming language is easy to define and can be found in several places (e.g. see [2]).

Then, the formal semantics of programs having been given, it is possible to define the notion of total correctness. P and Q being predicates on the stores, the total correctness formula $\{P\}\ p\ \{Q\}$ means "the evaluation of the program p on any store satisfying P terminates and the resulting store satisfies Q". Since the programming language is deterministic, there is at most one execution of a program and thus the total correctness formula may be written as:

$$\forall \sigma . P(\sigma) \Rightarrow \exists \tau . < p, \sigma > \rightarrow \tau \wedge Q(\tau)$$

To achieve the goal of verifying software, we must be able to check the validity of such correctness formulae. But such propositions are not easy to handle, since their definitions involve a semantic relation. Hoare logic solves this problem by introducing inference rules — the so-called Hoare rules — based on the syntax of programs. The rules corresponding to our language are given in figure 1. They are not only proved to be *sound* but also *complete*, assuming that we are able to establish the validity of propositions appearing in the consequence rule. Then we have no longer to reason about the semantic relation, but only about predicates on the stores.

$$\frac{}{\{P\}\ \mathsf{skip}\ \{P\}} \quad (\textsc{Skip}_{\mathcal{P}})$$

$$\frac{}{\{P[x \leftarrow t]\}\ x := t\ \{P\}} \quad (\textsc{Assign}_{\mathcal{P}})$$

$$\frac{\{P\}\ t_1\ \{R\} \quad \{R\}\ t_2\ \{Q\}}{\{P\}\ t_1 ; t_2\ \{Q\}} \quad (\textsc{Composition}_{\mathcal{P}})$$

$$\frac{\{P \wedge b = \mathsf{true}\}\ t_1\ \{Q\} \quad \{P \wedge b = \mathsf{false}\}\ t_2\ \{Q\}}{\{P\}\ \mathsf{if}\ b\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2\ \{Q\}} \quad (\textsc{Conditional}_{\mathcal{P}})$$

$$\frac{\{P \wedge b = \mathsf{true} \wedge \phi = z\}\ t\ \{P \wedge \phi < z\}}{\{P\}\ \mathsf{while}_{\phi}\ b\ \mathsf{do}\ t\ \mathsf{done}\ \{P \wedge b = \mathsf{false}\}} \quad (\textsc{Loop}_{\mathcal{P}})$$

$$\frac{P \Rightarrow P_1 \quad \{P_1\}\ t\ \{Q_1\} \quad Q_1 \Rightarrow Q}{\{P\}\ t\ \{Q\}} \quad (\textsc{Consequence}_{\mathcal{P}})$$

Figure 1: Hoare rules

1.2 Functional programs and realizability.

Things are easier in purely functional programming languages. Indeed, programs are now λ-terms, that are mathematical objects on which it is easy to reason and to compute. As a consequence, the semantic relation is now defined in terms of reduction (i.e. equality) of the program itself and the correctness formula becomes $\forall x.P(x) \Rightarrow \exists y.y = p(x) \wedge Q(y)$. Usually, we prefer to use a postcondition on both input and output, that is

$$\forall x.P(x) \Rightarrow \exists y.y = p(x) \wedge Q(x, y)$$

There are nowadays several implementations of theorem provers based on λ-calculi, in which such formulae can be formally proved, such as HOL, LEGO, Nuprl, etc. One of them is the system Coq [3], a Proof Assistant for the Calculus of Inductive Constructions [4, 14] (CIC for short).

Conversely, let π be a constructive proof of a proposition $S \equiv \forall x.P(x) \Rightarrow \exists y.Q(x,y)$. The notion of realizability [10] associates a program to that proof, which is its computational content. In that case, it is a program p computing the output y from the input x, and therefore π may be viewed as the proof of correctness of p. There are several ways of computing the realizer — the underlying program — and in the case of the CIC, this process is called *extraction* [13]. We shall denote the extracted program by $\mathcal{E}(\pi)$. We shall write $\Gamma \vdash P$ when P is provable under the assumptions Γ, and $\Gamma \vdash P\ [p]$ when the realizer is p.

Then, proving that a particular functional program p satisfies the specification S consists in constructing a proof π of S such that $\mathcal{E}(\pi) = p$ i.e. a proof of $\vdash S\ [p]$. Actually, it is possible to automatically construct some parts of π using p, in such a way that there only remain some logical goals, called the proof obligations. This methodology has been implemented in the Coq Proof Assistant and is called the Program tactic; it is described in [12].

1.3 How do they relate ?

Even though the first approach deals with imperative programs and the second with functional ones, they may be related. Indeed, let us interpret the total correctness formula $\{P\}\ p\ \{Q\}$ for an imperative program p, as the proposition $P(x) \vdash \exists y.Q(y)$ in the CIC, where x and y are tuples representing respectively the initial and the final stores of p. Then the Hoare rules given in figure 1 are valid for this interpretation. As a consequence, any proof of $\{P\}\ p\ \{Q\}$ in Hoare logic gives a proof π in the CIC of the proposition $P(x) \vdash \exists y.Q(y)$. And then the natural question is: *what is $\mathcal{E}(\pi)$?*

With our interpretation of the total correctness formula, $\mathcal{E}(\pi)$ is a functional term taking the input x of the program p and returning the output y: it is a *functional translation* of the imperative program p. Let us take a small example and see how it works.

Example 1 *Let us consider the program $p \equiv (y := y \times x\ ;\ x := x - 1)$ and the total correctness formula $\{P\}\ p\ \{Q\}$ where $P \equiv (x = 2 \wedge y = 2)$ and $Q \equiv (x = 1 \wedge y = 4)$. It can be derived using Hoare rules as follows, using the intermediate predicate $R \equiv (x = 2 \wedge y = 4)$*

$$\dfrac{\dfrac{\theta_1 \quad \overline{\{x = 2 \wedge y \times x = 4\}\ y := y \times x\ \{R\}}}{\{P\}\ y := y \times x\ \{R\}} \qquad \dfrac{\theta_2 \quad \overline{\{x - 1 = 1 \wedge y = 4\}\ x := x - 1\ \{Q\}}}{\{R\}\ x := x - 1\ \{Q\}}}{\{P\}\ p\ \{Q\}}$$

where θ_1 is the proposition $\forall(x,y).x = 2 \wedge y = 2 \Rightarrow x = 2 \wedge y \times x = 4$ and θ_2 the proposition $\forall(x,y).x = 2 \wedge y = 4 \Rightarrow x - 1 = 1 \wedge y = 4$.

Let us translate it into a constructive proof of $P(x,y) \vdash \exists(x',y').Q(x',y')$. The logic rules are given in appendix, figure 4. The only rules we need here are introduction and elimination of $\exists$, which are the following:

$$\dfrac{\Gamma \vdash L(t)}{\Gamma \vdash \exists x.L(x)\ [t]}(\exists\text{-}intro) \qquad \dfrac{\Gamma \vdash \exists x.L(x)\ [t] \quad \Gamma, L(x) \vdash P\ [e] \quad x \notin \Gamma, P}{\Gamma \vdash P\ [\mathsf{let}\ x = t\ \mathsf{in}\ e]}(\exists\text{-}elim)$$

Thus the previous deduction gives the proof

$$\dfrac{\dfrac{P(x,y) \vdash R(x, y\times x)}{P(x,y) \vdash \exists(x',y').R(x',y')\ [(x, y\times x)]} \quad \dfrac{P(x,y), R(x_1,y_1) \vdash Q(x_1-1, y_1)}{P(x,y), R(x_1,y_1) \vdash \exists(x',y').Q(x',y')\ [(x_1-1,y_1)]}}{P(x,y) \vdash \exists(x',y').Q(x',y')\ [\mathsf{let}\ (x_1,y_1) = (x, y\times x)\ \mathsf{in}\ (x_1-1, y_1)]}$$

So we get a program computing the new values of x and y which is $p(x,y) =$ let $x_1, y_1 = x, y \times x$ in $x_1 - 1, y_1$.

□

This example highlights two features. First, the extracted program is exactly the one we would have written "by hand", in the sense that it takes the values of the store necessary for the computation (x and y) and returns the values of the store modified by the computation (x' and y', as $x-1$ and $y \times x$). So it is closer to the mathematical meaning of p than usual representations in denotational semantics as *store transformers* that take the whole store and return the whole store, even when only few variables are read or written.

Secondly, it is much simpler to prove the correctness formula $P(x,y) \vdash \exists(x',y').Q(x',y')$ by giving the functional term $f \equiv$ let $x_1, y_1 = x, y \times x$ in $x_1 - 1, y_1$ and trying to construct of proof π such that $\mathcal{E}(\pi) = f$. Using the Program tactic, it remains to prove only one proof obligation:

$$x = 2 \wedge y = 2 \quad \Rightarrow \quad x - 1 = 1 \wedge y \times x = 4$$

Actually, it is exactly the computation of *weakest preconditions* in Hoare logic.

Weakest preconditions. Given a program p and a postcondition Q, there exists a proposition $wp(p)(Q)$, called the weakest precondition of p with respect to Q, such that $\{P\}\ p\ \{Q\}$ holds if and only if $P \Rightarrow wp(p)(Q)$ holds. In particular, we have $\{wp(p)(Q)\}\ p\ \{Q\}$.

In the general case, the proposition $wp(p)(Q)$ is not computable — this is a consequence of Gödel's incompleteness theorem. But it becomes computable for the fragment without loops. For instance, in example 1, we have that $wp(p)(Q) = x - 1 = 1 \wedge y \times x = 4$. Since proving the correctness formula is proving that the precondition implies the weakest precondition, we have to prove that $x = 2 \wedge y = 2 \quad \Rightarrow \quad x - 1 = 1 \wedge y \times x = 4$.

Auxiliary variables. It is often necessary to relate the values of variables at different moments of execution, typically before and after a sequence of instructions, and the solution is to use *auxiliary variables*. These are logical variables, distinct from the variables of the program (i.e. of the store), which are implicitly universally quantified in the correctness formulae.

For instance, when writing a specification of the factorial function one could write something like $\{\}\ p\ \{y = x!\}$, but the trivial program $x\ :=\ 1\ ;\ y\ :=\ 1$ will realize this specification. So one should write a correctness formula like $\{x = x_0\}\ p\ \{y = x_0!\}$ where x_0 is an auxiliary variable whose role is to relate the final value of y with the initial value of x.

Notice that auxiliary variables are *fresh* variables not appearing in the program, which are implicitly universally quantified. We shall illustrate the use of auxiliary variables in the example of the next paragraph.

Loops and recursion. Since we are interested in total correctness, we have to face the problem of the proof of termination of programs. In both formalisms, this proof is related to a well-founded order relation. In the case of imperative programs, the proof of termination is done by giving a quantity (which can be sometimes automatically determined) that strictly decreases for a well-founded order relation. Most often, this order is the usual order relation on natural numbers. In higher-order logics, and in particular in the CIC, we can define new order relations and prove that they are well-founded. Then we can prove propositions by well-founded inductions i.e. applying an induction principle of the kind

$$\forall P.(\forall x.(\forall y.y < x \Rightarrow P(y)) \Rightarrow P(x)) \Rightarrow \forall x.P(x) \tag{1}$$

To understand the relationship between loops in Hoare logic and recursion, let us prove the validity of the Hoare rule for loops when the correctness formula is interpreted by $P(x) \vdash \exists y.Q(y)$. We assume that the premise of ($\textsc{Loop}_{\mathcal{P}}$) is true i.e.

$$P(x), b(x) = \mathsf{true}, \phi(x) = z \vdash \exists y.(P(y) \wedge \phi(y) < z) \tag{2}$$

and we have to prove that $P(x) \vdash \exists y.(P(y) \wedge b(y) = \mathsf{false})$. To establish that fact, let us prove the strongest property $\forall \phi_0.\Theta(\phi_0)$, where

$$\Theta(\phi_0) \quad \equiv \quad \forall x.\phi(x) = \phi_0 \wedge P(x) \Rightarrow \exists y.(P(y) \wedge b(y) = \mathsf{false})$$

by well-founded induction on ϕ_0 and the result will follow by an instantiation of ϕ_0 by $\phi(x)$. A proof by well-founded induction corresponds to the rule with realizer

$$\frac{\Gamma, f : \forall x_1.x_1 < x \Rightarrow Q(x_1) \vdash Q(x)\ [e]}{\Gamma \vdash Q(x)\ [\mathsf{let\ rec}\ f\ x = e\ \mathsf{in}\ f\ x]}$$

Therefore we have to establish that $f : IH, \phi(x) = \phi_0, P(x) \vdash \exists y.(P(y) \wedge b(y) = \mathsf{false})$ where

$$IH \quad \equiv \quad \forall \phi_1.\phi_1 < \phi_0 \Rightarrow \forall x.\phi(x) = \phi_1 \Rightarrow P(x) \Rightarrow \exists y.(P(y) \wedge b(y) = \mathsf{false})$$

At this step, we reason by case on the value of $b(x)$, which corresponds to the rule

$$\frac{\Gamma, b = \mathsf{true} \vdash Q\ [e_1] \quad \Gamma, b = \mathsf{false} \vdash Q\ [e_2]}{\Gamma \vdash Q\ [\mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]}$$

The case of the right premise ($b(x) = \mathsf{false}$) is easy: we just have to take $y = x$. In the other case ($b(x) = \mathsf{true}$) we use the hypothesis (2) with $z = \phi_0$ and we get an x_1 such that $P(x_1) \wedge \phi(x_1) < \phi_0$ holds. Then we can apply the induction hypothesis IH on $\phi_1 = \phi(x_1)$ and $x = x_1$ and the result holds. □

Putting all together, the realizer associated to the derivation of $\{P\}$ $\mathsf{while}_\phi\ b\ \mathsf{do}\ t\ \mathsf{done}$ $\{P \wedge b = \mathsf{false}\}$ is the program

$$\mathsf{let\ rec}\ f\ x = \mathsf{if}\ b\ \mathsf{then\ let}\ x_1 = e(x)\ \mathsf{in}\ f\ x_1\ \mathsf{else}\ x\ \mathsf{in}\ f\ x$$

where $e(x)$ is the realizer associated to the derivation of $\{P \wedge b = \mathsf{true} \wedge \phi = z\}\ t\ \{P \wedge \phi < z\}$ i.e. to the body of the loop. Notice that this recursive function expresses an unfolding of the loop which is traditionally written as the following equivalence:

$$\mathsf{while}\ b\ \mathsf{do}\ t\ \mathsf{done} \quad \approx \quad \mathsf{if}\ b\ \mathsf{then}\ (t\ ;\ \mathsf{while}\ b\ \mathsf{do}\ t\ \mathsf{done})\ \mathsf{else\ skip}$$

Let us illustrate this relationship between loops and recursion on an example.

Example 2 *Let us consider the factorial function. We choose the following implementation*

$$p \quad \equiv \quad y := 1 \; ; \; \mathsf{while}\; x > 0 \;\mathsf{do}\; y := y \times x \; ; \; x := x - 1 \;\mathsf{done}$$

and we wish to prove the following correctness formula

$$S \quad \equiv \quad \{x = x_0 \wedge x \geq 0\}\; p \;\{y = x_0!\}$$

The derivation of the correctness proof is quite lengthy and so we present it in a sequential manner, omitting the trivial steps:

$\{x = x_0 \wedge x \geq 0\}$	
$y := 1$;	
$\{y = 1 \wedge x = x_0 \wedge x \geq 0\}$	Assign + Consequence
$\{x! \times y = x_0! \wedge x \geq 0\}$	Consequence
$\mathsf{while}_x\; x > 0 \;\mathsf{do}$	
$\quad \{x! \times y = x_0! \wedge x \geq 0 \wedge x > 0 \wedge x = z\}$	
$\quad y := y \times x$;	
$\quad x := x - 1$	
$\quad \{x! \times y = x_0! \wedge x \geq 0 \wedge x < z\}$	2 × Assign + Consequence
done	
$\{x! \times y = x_0! \wedge x \geq 0 \wedge x \leq 0\}$	Loop
$\{y = x_0!\}$	Consequence

As for example 1, this derivation in Hoare deduction calculus can be translated into a constructive proof in the CIC of the proposition $x = x_0 \wedge x \geq 0 \vdash \exists(x', y').y' = x_0!$. *Let* π *be that proof. Then, after having reduced some* let in *constructs, we get*

$$\mathcal{E}(\pi)(x, y) \quad = \quad \mathsf{let\ rec}\; f\ (x, y) = \mathsf{if}\; x > 0 \;\mathsf{then}\; f\ (x - 1, y \times x) \;\mathsf{else}\; (x, y) \;\mathsf{in}\; f\ (x, 1)$$

We can see that $\mathcal{E}(\pi)$ *is a function computing the new values of* x *and* y *from their initial values, very close to the usual way to write the factorial function (except that, in that case, we have an extra argument* y *and an extra result* x*).*

□

2 Program correctness in Type Theory

Following the ideas developed in the previous section, we would like to mix features from Hoare style and type theoretic frameworks to get an improved methodology for showing correctness of imperative programs. We keep the same small imperative language for the moment.

As we explained before, our main purpose is the possibility to express the correctness formula in the same logical language as specifications, and not only in a meta-level logical language as it is usually done. Then, the correctness formula being a proposition fully expressible in the logic, we can *prove it as we want*, using the full expressiveness and power of higher order. Of course, we shall also give a methodology similar to the Hoare deduction rules to automate a large part of correctness proofs.

Before-after predicates. Firstly, an obvious requirement is the ability to use before-after predicates in postconditions, i.e. to speak of the values of variables before and after the computation. Indeed, in the classical correctness formulae, we can only speak of the values *after* the computation, and this restriction implies a huge use of auxiliary variables, as we illustrated in the previous section. For instance, the specification languages of VDM and Z both provide a way to refer to the old values of objects.

Let V be the set of the variables of the store. Let V' be a copy of V that belongs to a distinct syntactic class; say, for instance, that the variables of V' are written with a ' and those of V are not. Then, a *precondition* is a predicate over the variables of V and a *postcondition* is a predicate over the variables of V and V'. The variables of V' represent the values of the variables *after* the computation. For instance, the specification of the factorial function will become $\{\}\ p\ \{y' = x!\}$.

Correctness formulae expressed in type theory. We explained that we prefer a correctness formula that we can fully express in our logical framework, in such a way that we can handle it and prove it as we want. So, instead of having a semantic (mathematical) definition of the proposition *"the program p evaluated on the initial store σ terminates on the store τ"*, we will prefer to express it directly *as a computation*. To achieve this goal, we consider a functional translation of the imperative program p, that is a function taking the input of the program and returning its output. But instead of taking and returning a whole store, as in denotational semantics, we will consider a functional program which takes only the values which are necessary for the computation and returns the minimal finite set of values (possibly) assigned by the program.

Such a functional translation for the small imperative programs we are considering here is easy to define. It is a particular case of more general ways to translate imperative programs into functional ones. For instance, P.W. O'Hearn and J. C. Reynolds recently described how to translate Algol programs into a purely functional language, in an unpublished article [11]. Independently, we introduced another way to do such translations based on monads; this work is described in [5]. Without entering into technical details, let us briefly describe what we mean by this functional translation. Let p be a program and $X_p = \mathsf{var}(p)$ the set of its variables. Then a functional translation of p is a functional term $\overline{p}$ of type $\mathsf{int}^{X_p} \rightarrow \mathsf{int}^{X_p}$, where int^{X_p} is the space of functions from X_p to int. $\overline{p}$ is assumed to have the same semantics as p i.e. for all stores σ and τ, of type int^V, we have

$$< p, \sigma > \rightarrow \tau \qquad \Longleftrightarrow \qquad \forall y.\ \tau(y) = \begin{cases} \overline{p}(\sigma_{|X_p})(y) & \text{if } y \in X_p \\ \sigma(y) & \text{if } y \notin X_p \end{cases}$$

where $\sigma_{|X_p}$ is the restriction of σ to the domain X_p. If X is a set of variables including X_p we denote by $\overline{p}_X$ the canonical extension of $\overline{p}$ to a function of type $\mathsf{int}^X \rightarrow \mathsf{int}^X$ (i.e. such that $\overline{p}_X(f)(x) = f(x)$ if $x \notin X_p$ and $\overline{p}_X(f)(x) = \overline{p}(f_{|X_p})(x)$ otherwise).

Now, let us define the correctness formula. A precondition is a predicate P over some variables of V and a postcondition is a predicate Q over some variables of V and some variables of V'. Then let X be the union of all these variables — taken in V — and of the variables of $\mathsf{var}(p)$. Let A be the set of auxiliary variables appearing in both P and Q. Then the interpretation of the correctness formula $\{P\}\ p\ \{Q\}$ is defined as

$$[\![\{P\}\ p\ \{Q\}]\!] \quad \stackrel{\text{def}}{=} \quad P(X) \Rightarrow \exists X'.X' = \overline{p}_X(X) \wedge Q(X, X') \tag{3}$$

where X and X' are sets of variables of type int in this formula, but considered as functions of type int^{X_p} in the equality $X' = \overline{p}_X(X)$ in order to simplify the notation. Notice that the variables of X and A are free in this proposition.

NOTE: It is possible to define this interpretation using $\overline{p}$ instead of $\overline{p}_X$, expressing this way the fact that some variables of V' in Q are actually not modified by p, but we chose this formulation here in an attempt to simplify the presentation.

Proof system. We now give a proof system for the new notion of correctness formulae. This system, called $\mathcal{F}$, is given in figure 2. We write $\vdash_{\mathcal{F}} \{P\}\ p\ \{Q\}$ when the correctness formula $\{P\}\ p\ \{Q\}$ is derivable using $\mathcal{F}$. These rules need some comments. The rules for

$$\frac{}{\{Q(X,X)\}\ \mathsf{skip}\ \{Q(X,X')\}} \qquad (\textsc{Skip}_{\mathcal{F}})$$

$$\frac{}{\{Q(X,X[x \leftarrow t])\}\ x := t\ \{Q\}} \qquad (\textsc{Assign}_{\mathcal{F}})$$

$$\frac{\{P(X)\}\ t_1\ \{R(X,X')\} \quad \{R(X_i,X)\}\ t_2\ \{Q(X_i,X')\}}{\{P(X)\}\ t_1; t_2\ \{Q(X,X')\}} \qquad (\textsc{Composition}_{\mathcal{F}})$$

$$\frac{\{P(X) \wedge b = \mathsf{true}\}\ t_1\ \{Q(X,X')\} \quad \{P(X) \wedge b = \mathsf{false}\}\ t_2\ \{Q(X,X')\}}{\{P(X)\}\ \mathsf{if}\ b\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2\ \{Q(X,X')\}} \qquad (\textsc{Conditional}_{\mathcal{F}})$$

$$\frac{\{Q(X_i,X) \wedge b = \mathsf{true}\}\ t\ \{Q(X_i,X') \wedge \phi[X \leftarrow X'] < \phi\}}{\{Q(X,X)\}\ \mathsf{while}_\phi\ b\ \mathsf{do}\ t\ \mathsf{done}\ \{Q(X,X') \wedge b[X \leftarrow X'] = \mathsf{false}\}} \qquad (\textsc{Loop}_{\mathcal{F}})$$

$$\frac{P \Rightarrow P_1 \quad \{P_1(X)\}\ t\ \{Q_1(X,X')\} \quad Q_1 \Rightarrow Q}{\{P(X)\}\ t\ \{Q(X,X')\}} \qquad (\textsc{Consequence}_{\mathcal{F}})$$

Figure 2: new deduction rules (to establish $\vdash_{\mathcal{F}} \{P\}\ p\ \{Q\}$)

skip, assignment, conditional and consequence are somewhat similar to the traditional ones and are easy to understand. In the rule for composition, some fresh auxiliary variables X_i are introduced in the right premise. They represent the values of the variables before the evaluation of the sequence $t_1\ ;\ t_2$, whereas X in the right premise would have refered to the values before the evaluation of t_2 i.e. in the intermediate state of the sequence. Similarly, the auxiliary variables X_i in the rule for loop represent the values of the variables before the evaluation of the whole loop, while X and X' in the premise refer to the values before and after one evaluation of the body t of the loop.

As for the traditional deduction system of Hoare rules, we have the following results.

Proposition 1 (Soundness) *The proof system $\mathcal{F}$ is sound, i.e.*

$$\vdash_{\mathcal{F}} \{P\}\ p\ \{Q\} \quad \Rightarrow \quad [\![\{P\}\ p\ \{Q\}]\!] \textit{ is true}$$

PROOF OUTLINE. The proof is straightforward. For each deduction rule, we have to prove that the conclusion is a consequence of the premises. The only subtle case is for the loop, where we have to apply a well-founded induction principle. $\square$

Proposition 2 (Completeness) *The proof system $\mathcal{F}$ is complete, i.e.*

$$[\![\{P\}\ p\ \{Q\}]\!] \text{ is true} \qquad \Rightarrow \qquad \vdash_{\mathcal{F}} \{P\}\ p\ \{Q\}$$

PROOF OUTLINE. The proof is quite standard, following traditional ones as in [2]. We first introduce a notion of *weakest precondition* such that $\{P\}\ p\ \{Q\}$ holds if and only if $P \Rightarrow wp(p)(Q)$ holds. Here, the weakest precondition is directly defined as

$$wp(p)(Q) \quad \stackrel{\text{def}}{=} \quad \exists X'.X' = \overline{p}_X(X) \wedge Q(X, X') \tag{4}$$

Then it is only necessary to prove that $\vdash_{\mathcal{F}} \{wp(p)(Q)\}\ p\ \{Q\}$ and the result will follow using the consequence rule. To establish this fact, we prove some properties of the weakest precondition.

Proposition 3 *The weakest precondition satisfies the following properties:*

(1) $wp(\mathsf{skip})(Q(X, X')) \Leftrightarrow Q(X, X)$

(2) $wp(x := t)(Q(X, X')) \Leftrightarrow Q(X, X[x \leftarrow t])$

(3) $wp(t_1; t_2)(Q(X, X')) \Leftrightarrow wp(t_1)((wp(t_2)(Q(X_i, X')))[X \leftarrow X'][X_i \leftarrow X])$

(4) $wp(\mathsf{if}\ b\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2)(Q(X, X'))$
$\qquad \Leftrightarrow \mathsf{if}\ b\ \mathsf{then}\ wp(t_1)(Q(X, X'))\ \mathsf{else}\ wp(t_2)(Q(X, X'))$

(5) $wp(\mathsf{while}_\phi\ b\ \mathsf{do}\ t\ \mathsf{done})(Q(X_i, X')) \wedge b = \mathsf{true}$
$\qquad \Rightarrow wp(t)((wp(\mathsf{while}_\phi\ b\ \mathsf{do}\ t\ \mathsf{done})(Q(X_i, X')))[X \leftarrow X'])$

(6) $(wp(\mathsf{while}_\phi\ b\ \mathsf{do}\ t\ \mathsf{done})(Q(X_i, X')))[X \leftarrow X'] \wedge b[X \leftarrow X'] = \mathsf{false} \Rightarrow Q(X_i, X')$

Notice that those properties of wp allow us, for the fragment without loops, to compute it recursively from the structure of the program.

$\square$

These results show how it is possible the give a precise definition of the total correctness formula in presence of before-after predicates and auxiliary variables, a correct and complete deduction calculus "à la Hoare" being still definable. It now becomes important to move a step further and to study the case of a more realistic programming language.

3 A logic for real programming languages

3.1 The programming language Real.

Let us consider now a more powerful programming language, called Real, that mixes imperative and functional features. On one hand, in contrast with purely imperative programming languages, a program is no longer a sequence of commands, but is now an *expression* of atomic type, and we have now functions (procedures are just functions returning a value of type unit). On the other hand, in contrast with purely functional languages, we still have references, sequences and loops.

Atomic types (A) are the type bool of booleans, the type int of integers and the type unit of commands. Base types (B) are either atomic types or types of references on integers, written int ref. Access to the value of the reference x is written $!x$. Functions have types of the form $B_1 \to \cdots \to B_n \to A$, which means that functions take either arguments by values or by references and return values of atomic types. To simplify the presentation, we assume that arguments passed by values are given first and then those given by references. We do not consider here the case of partial applications. Notice that the presence of local references (let ref) allow us to have local variables in functions.

Programs are closed expressions that follow the syntax given in figure 3 and that are correctly typed with respect to the typing rules given in the appendix, figure 5.

$$
\begin{array}{rcll}
M & ::= & v & \text{values} \\
 & | & x & \text{variable} \\
 & | & !x & \text{access} \\
 & | & x := M & \text{store} \\
 & | & M \ ; \ M & \text{sequence} \\
 & | & \mathsf{if}\ M\ \mathsf{then}\ M\ \mathsf{else}\ M & \text{conditional} \\
 & | & \mathsf{while}_\phi\ M\ \mathsf{do}\ M\ \mathsf{done} & \text{loop} \\
 & | & \mathsf{let}\ x = \mathsf{ref}\ M\ \mathsf{in}\ M & \text{new reference} \\
 & | & (op\ M \ldots M) & \text{app. of primitive operation} \\
 & | & ([x:B]\ldots[x:B]M\ M \ldots M) & \text{app. of function}
\end{array}
$$

Figure 3: Syntax of Real

3.2 Correctness formulae.

Since programs are now expressions, we have to extend the notion of postcondition to establish properties of *the result of programs*. For this purpose, a postcondition will now be a predicate over the variables of V and V', and also over a special variable r that stands for the result of the program. Moreover, the functional translation of a program p of type A is now a term $\overline{p}$ of type $\mathsf{int}^{X_p} \to \mathsf{int}^{X_p} \times A$, i.e. a term that takes the values of the variables used by p and that returns the new values of these variables together with the result of p.

Then we can define the correctness formula, which is somewhat similar to the correctness formula (3) defined in the previous section. With the same notations as before, we define

$$\{P\}\ p\ \{Q\} \quad \stackrel{\text{def}}{=} \quad P(X) \Rightarrow \exists(X',r).(X',r) = \overline{p}_X(X) \wedge Q(X,X',r) \tag{5}$$

Notice that the value of the reference x is written $!x$ inside programs to avoid confusion with the reference itself, but is directly written x in the logical propositions.

3.3 Deduction rules.

The new deduction rules to establish correctness formulae are given in appendix, figure 6 on page 20. They differ quite a lot from the classical Hoare rules, mainly because expressions can now cause side effects and must be treated as programs. See for instance the rule (store) of assignment which is now completely different from the rule ($\textsc{assign}_{\mathcal{F}}$) given in figure 2 page 9. The rule for the loop has also changed. The idea is still to prove that an invariant holds during the whole execution of the loop, but the difference is now that the test b is any expression of type bool, and may cause some side-effects. Therefore, instead of just writing $b = \mathsf{true}$ at the entrance of the loop and $b = \mathsf{false}$ at its exit, we can use any predicate R. If b is purely functional we can choose for R the predicate $b = \mathsf{r}$; then the first premise becomes trivially true and we find again the same rule as before (see figure 2).

The rules for application may seem complicated because they are given in their full generality, for any arity. (Notice that we chose to evaluate arguments of functions from left to right. We could have chosen to do the converse, but in presence of side-effects it would have given a completely different semantics to our language). These rules are easy to understand and quite natural when instantiated on small arities. Let us give some examples.

Unary operation. For a primitive unary operation op we get the deduction rule

$$\frac{\{P(X)\}\ e\ \{Q(X, X', (op\ \mathsf{r})\}}{\{P(X)\}\ (op\ e)\ \{Q(X, X', \mathsf{r})\}}$$

Binary operation. For a primitive binary operation op we get the deduction rule

$$\frac{\{P(X)\}\ e_1\ \{R(X, X', \mathsf{r}\} \qquad \{R(X_0, X, v_1\}\ e_2\ \{Q(X_0, X', (op\ v_1\ \mathsf{r})\}}{\{P(X)\}\ (op\ e_1\ e_2)\ \{Q(X, X', \mathsf{r})\}}$$

Function application. Let us consider the case of a function taking two arguments, the first one by value, of type int, and the second by reference, of type $\mathsf{int\ ref}$. Then the deduction rule is

$$\frac{\{P(X)\}\ e_1\ \{R(X, X', \mathsf{r})\} \qquad \{R(X_0, X, x)[y \leftarrow z]\}\ e\ \{Q(X_0, X', \mathsf{r})[y \leftarrow z]\}}{\{P(X)\}\ ([x : \mathsf{int}][z : \mathsf{int\ ref}]e\ e_1\ z)\ \{Q(X, X', \mathsf{r})\}}$$

where $[y \leftarrow z]$ stands for the substitution of y by z in X, X' and X_0. In the right premise, the auxiliary variables X_0 represent the values of variables before the whole execution of the β-redex; this is similar to what is done in the rule for composition, and this will be justified in the next paragraph.

Let us give some examples of correctness proofs with this new system.

Example 3 *First, let us consider a trivial correctness proof. Let p be the program $x := !x+1$, without any precondition and with the postcondition $Q \equiv x' > x$. The deduction is the following*

$$\dfrac{\dfrac{\dfrac{x+1>x \qquad \dfrac{}{\{x+1>x\}\ !x\ \{\mathsf{r}+1>x\}}}{\{\}\ !x\ \{\mathsf{r}+1>x\}} \qquad \dfrac{}{\{v_1+1>x_0\}\ 1\ \{v_1+\mathsf{r}>x_0\}}}{\{\}\ !x+1\ \{\mathsf{r}>x\}}}{\{\}\ x := !x+1\ \{x'>x\}}$$

and so the only logical premise to prove is

$$x + 1 > x$$

$\square$

Example 4 *Then, let us illustrate how it works with a function application. Let f be the function that augments a reference with a given value, that is*

$$f \equiv [x : \mathsf{int}][y : \mathsf{int\ ref}]y := !y + x$$

Let p be the program $(f\ 3\ z)$ with the precondition $P \equiv z > 0$ and the postcondition $Q \equiv z' > 3$. The derivation is the following

$$\dfrac{\mathcal{D} \qquad \dfrac{\dfrac{\overline{\{y + x > 3\}\ !y\ \{\mathsf{r} + x > 3\}} \qquad \overline{\{v_1 + x > 3\}\ x\ \{v_1 + \mathsf{r} > 3\}}}{\{y + x > 3\}\ !y + x\ \{\mathsf{r} > 3\}}}{\{y + x > 3\}\ y := !y + x\ \{y' > 3\}}}{\{z > 0\}\ (f\ 3\ z)\ \{z' > 3\}}$$

where $\mathcal{D}$ is the derivation

$$\dfrac{z > 0 \Rightarrow z + 3 > 3 \qquad \overline{\{z + 3 > 3\}\ 3\ \{z + \mathsf{r} > 3\}}}{\{z > 0\}\ 3\ \{z + \mathsf{r} > 3\}}$$

and so the only logical premise to prove is

$$z > 0 \Rightarrow z + 3 > 3$$

$\square$

Application as a let in construct. It is important to notice that the rule for function application is not really a rule of β-reduction, since there is no real substitution of the formal arguments by the real ones. It is better to see it as a sequence of bindings of several values in an expression i.e. as a sequence of let in constructs. Indeed, a β-redex may be rewritten like this

$$([x_1 : A_1] \dots [x_k : A_k]e\ e_1\ \dots\ e_k) \equiv \mathsf{let}\ x_1 = e_1\ \mathsf{in}\ \mathsf{let}\ x_2 = e_2\ \mathsf{in}\ \dots \mathsf{let}\ x_k = e_k\ \mathsf{in}\ e$$

By the way, we could add the construct let $x = M$ in M to the syntax of our language Real, and the corresponding deduction rule would be

$$\dfrac{\{P(X)\}\ e\ \{R(X, X', \mathsf{r})\} \qquad \{R(X_0, X, x)\}\ e'\ \{Q(X_0, X', \mathsf{r})\}}{\{P(X)\}\ \mathsf{let}\ x = e\ \mathsf{in}\ e'\ \{Q(X, X', \mathsf{r})\}}$$

Using this deduction rule for each argument e_i of the function, and some substitutions for the arguments which are references, we find again exactly the same rule as the rule (β) given in figure 6, page 20.

Actually, it is possible to consider a $\mathsf{let\ in}$ construct as a sequence, by introducing a new reference variable. Indeed, we can write $\mathsf{let}\ x = e\ \mathsf{in}\ e' \equiv x := e\ ;\ e'[x \leftarrow !x]$. Then, a function application can be considered as a sequence of assignments followed by the body of the function i.e.

$$([x_1 : A_1] \ldots [x_k : A_k]e\ e_1\ \ldots\ e_k) \equiv x_1 := e_1\ ;\ \ldots\ ;\ x_k := e_k\ ;\ e[x_i \leftarrow !x_i]$$

with still some substitutions for the references given as arguments. Seen like this, the rule for function application becomes really obvious.

But it is clearly not the good way to handle functions and that is the problem we shall consider in the next section.

4 Structured programming and recursive functions

Until now we have considered a program as a single closed term. It is clear that this is not the case in practice and that programs are split into some more or less elementary functions. And so must be the correctness proofs. The idea is to associate a correctness formula $\{P_f\}\ f\ \{Q_f\}$ to the definition of each function $f = [\vec{x}][\vec{z}]e$. To prove it is just proving it for the body e. It is expressed by the following rule

$$\frac{\{P_f(X)\}\ e\ \{Q_f(X, X', \mathsf{r})\}}{\{P_f(X)\}\ (f\ x_1\ \ldots\ x_k\ z_1\ \ldots\ z_n)\ \{Q_f(X, X', \mathsf{r})\}}\ (\textsc{Abstraction})$$

Once the correctness formula for f is proved, or assumed, it can be used to do other correctness proofs and it should not be necessary to look again at the body of f. So correctness proofs are now done in environments of the kind

$$\Gamma \quad ::= \quad \emptyset \ \mid\ \Gamma,\ \forall A.\forall X.\{P_f(X)\}\ f\ \{Q_f(X, X', \mathsf{r})\}$$

where A stands for the auxiliary variables appearing in the correctness formula of f. The variables of A and those of X must be abstracted in the correctness formula since this one may be used in different contexts.

Then, one could think that the consequence rule is exactly the rule we need to use informations of the context, but that is not so. Indeed, suppose for instance that we have specified a function f that augments a reference with a given value i.e. we assume the correctness formula $\{\}\ (f\ x\ y)\ \{y' = y + x\}$ to be in the context. Then we want to use this assumption to prove the correctness formula $\{z = 0\}\ (f\ 1\ r)\ \{r' = r + 1 \wedge z' = 0\}$. Omitting the premise corresponding to the evaluation of the first argument, 1, an application of the consequence rule would give us the premises:

$$\frac{z = 0 \Rightarrow \mathsf{True} \qquad \{\}\ (f\ 1\ r)\ \{r' = r + 1\} \qquad r' = r + 1 \Rightarrow r' = r + 1 \wedge z' = 0}{\{z = 0\}\ (f\ 1\ r)\ \{r' = r + 1 \wedge z' = 0\}}$$

and clearly the third one is not provable. Indeed, two main facts are not expressed by the consequence rule: firstly that $z = 0$ should still be available to establish the postcondition, and secondly that z is not modified by f (so that we can replace z' by z).

Actually, each of these two problems can be easily solved. Firstly, the fact that the precondition still holds after the computation — as a predicate of the variables representing the old values, of course — is expressed by the following rule:

$$\frac{\{P(X)\}\ e\ \{Q(X,X',\mathsf{r})\}}{\{P(X)\}\ e\ \{P(X) \wedge Q(X,X',\mathsf{r})\}}\quad (\textsc{Persistence})$$

which is clearly sound.

Secondly, the fact that some variables are not modified by a program is expressed by the following rule:

$$\frac{\{P(X)\}\ e\ \{Q(X,Y,X',\mathsf{r})\} \qquad Y \cap \mathsf{var}(e) = \emptyset}{\{P(X)\}\ e\ \{Q(X,Y',X',\mathsf{r})\}}\quad (\textsc{Identity})$$

which is also sound since $\overline{e}_X$ is the identity on the variables of X that do not belong to $\mathsf{var}(e)$.

Function application. Since the two previous rules can be used anywhere, it is difficult to use them significantly in an automatic application of the deduction rules. But we can actually restrict their use to function application. The rule we propose for application (app) is given in appendix, figure 6. Let us illustrate it on the example of a function taking two arguments, one by value and one by reference. The corresponding rule is:

$$\frac{\begin{array}{c}\{P(X)\}\ e\ \{R(X,X',\mathsf{r})\} \qquad R(X_0,X,x) \Rightarrow P_f(X)[z \leftarrow y] \\ R(X_0,X,x) \wedge Q_f(X,X',\mathsf{r})[z \leftarrow y] \Rightarrow Q(X_0,Y,W',\mathsf{r}) \qquad Y \cap \mathsf{var}(f) = \emptyset\end{array}}{\{P(X)\}\ (f\ e\ y)\ \{Q(X,Y',W',\mathsf{r})\}}$$

under the assumption that a correctness formula for f of the kind $\{P_f\}\ (f\ x\ z)\ \{Q_f(X,X',\mathsf{r})\}$ belongs to the context. This rule illustrates the fact that we first evaluate the argument e, leading to a predicate R, then we have to prove that the precondition P_f of the function is true under the assumption R and finally we have to establish the postcondition Q under the assumptions R and Q_f.

Recursive functions. We are now in position to deal with recursive functions. As do the loops, the recursive functions also carry an argument of well-founded induction, as a quantity ϕ. So recursive functions will be written $\mathsf{Rec}_\phi\ f = [x_1]\ldots[x_n]e$. Since we have now a rule for function application, the only thing to do is to give a rule to establish the correctness of a recursive function. The idea is to prove the correctness formula *under the assumption that it holds for smaller calls of the function*, in the sense of the well-founded induction.

So, writing explicitly the context Γ in which we do the correctness proof, the rule to derive the correctness formula for a recursive function $F \equiv \mathsf{Rec}_\phi\ f = [\vec{x}]e$ is the following:

$$\frac{\Gamma, \forall Y.\{P(Y) \wedge \phi(Y) < \phi(X)\}\ (f\ \vec{y})\ \{Q(Y,Y',\mathsf{r})\} \vdash \{P(X)\}\ e\ \{Q(X,X',\mathsf{r})\}}{\Gamma \vdash \{P(X)\}\ (F\ \vec{x})\ \{Q(X,X',\mathsf{r})\}}\quad (\textsc{Rec})$$

We have shown in this section that it is possible to keep the structure of programs when doing correctness proofs, by associating a correctness formula to each function. This way, it enables *modularity* in correctness proofs, in the sense that it is possible to assume and to use the specification of a function without having to implement it, which is crucial in real software validation.

5 Conclusion and future work

Two main ideas summarize what we have presented in this paper. First, we have proposed a correctness formula for imperative programs in Type Theory. The main advantage is that there are several robust implementations of theorem provers for type theoretic frameworks (HOL, Coq, PVS, etc.). Moreover, they are general provers i.e. in which we can define new notions and prove new theorems. This is not the case in specialized provers for one particular logic.

Let us compare our approach to the approach of the B method [1]. There are not so many differences in the proof obligations, even if types allow not to consider proof obligations of the kind $t \in \mathsf{int}$ since they are treated by the decidable typing judgment. But the way the proof obligations are generated and proved are really different. Indeed, in the case of the B method, the proof obligations are generated from the specifications by an external program, the proof obligations generator, and passed to another program, the prover, which tries to prove them using a huge database of logic rules (more than 2000). In case of failure, it is possible to add unjustified axioms in the database of the prover. The specifications and the proof obligations do not belong to the same logic; actually, the correctness formula is not even expressed.

In our approach, on the contrary, the correctness formula is expressed in the same logic as the specifications. The generation of the proof obligations is now just a particular tactic to help the user in proving the correctness formulae. Therefore, if the user need more notions and more theorems to fulfill its proof (a proof of well-foundness for instance) he can use the all power of the theorem prover to do so.

The second main result of this paper is the extension of what was traditionally done for imperative programming languages with functional features, still keeping a set of Hoare deduction rules and a notion of weakest precondition. Then it was rather easy to give sound deduction rules for functions, even recursive ones. In the way, this proof of correctness of imperative programs is no longer restricted to imperative programming languages like C, Pascal or ADA, but can be applied to functional languages with imperative features, like SML or Objective Caml, which has never been done previously to our knowledge.

A step further. A lot of work is currently in progress to get a real environment for program validation in the Calculus of Inductive Constructions. Firstly, we must define a functional translation of imperative programs, which is necessary to define the correctness formula. This translation has to be proved correct with respect to the semantics of the programming language. This is described in a forthcoming paper [5].

But this alone is not enough, of course. We would like to add two main aspects to the programming language, which are data-types and exceptions. Concerning data-types, the case of arrays or tuples is quite easy to handle but the general case of recursive data-types — with mutable parts — is not. But this is necessary to prove real programs dealing with lists, trees, etc. Exceptions are also a fundamental aspect in real software development and they have to be understood on the point of view of correctness. This work is still in progress.

Acknowledgments. I would like to thank first of all Christine Paulin, my supervisor, not only for her help but also for her trust and her patience during the long and difficult genesis of this work. I am also grateful to both Judicaël Courant and Hugo Herbelin for remarks and discussions about program validation. Finally, I wish to thank Ajay Chander for a very detailed reading of this paper.

References

[1] J. R. Abrial. *The B-Book. Assigning programs to meaning.* Cambridge University Press, 1996.

[2] K. R. Apt and E.-R. Olderog. *Verification of sequential and concurrent programs.* Springer-Verlag, 1991.

[3] B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi, and B. Werner. *The Coq Proof Assistant Reference Manual Version 6.1*, December 1996.

[4] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3), 1988.

[5] J.-C. Filliâtre. Functional translation of imperative programs. To appear.

[6] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types.* Cambridge University Press, 1989.

[7] I. Hayes. *Specification Case Studies.* Oxford University Computing Laboratory, 1985. Technical monograph PRG-46.

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

[9] C. B. Jones. *Systematic Software Development using VDM.* C. A. R. Hoare Series Editor. Prentice Hall, 1989.

[10] S.C. Kleene. *Introduction to Metamathematics.* Bibliotheca Mathematica. North-Holland, 1952.

[11] P. W. O'Hearn and J. C. Reynolds. From Algol to Polymorphic Linear Lambda-calculus. April 1997.

[12] 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.

[13] 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.

[14] C. Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In M. Bezem and J.-F. Groote, editors, *Proceedings of the conference Typed Lambda Calculi and Applications*, number 664 in LNCS, 1993. Also LIP research report 92-49.

[15] W. Reif. The KIV-approach to Software Verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report*. Springer LNCS 1009, 1995.

[16] 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.

Logical propositions:	L		
Informative propositions:	P	$::=$	$L \mid L \Rightarrow P \mid \forall x.P \mid \exists x.L$
Realizers:	e	$::=$	$t \mid \lambda x.e \mid (e\ e) \mid \mathsf{if}\ t\ \mathsf{then}\ e\ \mathsf{else}\ e$
		$\mid$	$\mathsf{let}\ x = e\ \mathsf{in}\ e \mid \mathsf{let\ rec}\ x = e\ \mathsf{in}\ e \quad (t \text{ is a term})$
Proofs environments:	Γ	$::=$	$\emptyset \mid \Gamma, L \mid \Gamma, x : P$

$$\frac{\Gamma \vdash L(t)}{\Gamma \vdash \exists x.L(x)\ [t]} \qquad \frac{\Gamma \vdash \exists x.L(x)\ [t] \quad \Gamma, L(x) \vdash P\ [e] \quad x \notin \Gamma, P}{\Gamma \vdash P\ [\mathsf{let}\ x = t\ \mathsf{in}\ e]}$$

$$\frac{\Gamma \vdash P(x)\ [e] \quad x \notin \Gamma}{\Gamma \vdash \forall x.P(x)\ [\lambda x.e]} \qquad \frac{\Gamma \vdash \forall x.P(x)\ [e]}{\Gamma \vdash P(t)\ [(e\ t)]} \qquad \frac{\Gamma, L \vdash P\ [e]}{\Gamma \vdash L \Rightarrow P\ [e]} \qquad \frac{\Gamma \vdash L \Rightarrow P\ [e] \quad \Gamma \vdash L}{\Gamma \vdash P\ [e]}$$

$$\frac{\Gamma, b = \mathsf{true} \vdash P\ [e_1] \quad \Gamma, b = \mathsf{false} \vdash P\ [e_2]}{\Gamma \vdash P\ [\mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]}$$

$$\frac{\Gamma, f : \forall x_1.x_1 < x \Rightarrow P(x_1) \vdash P(x)\ [e]}{\Gamma \vdash P(x)\ [\mathsf{let\ rec}\ f\ x = e\ \mathsf{in}\ f\ x]}$$

Figure 4: Logic rules

Atomic types:	A	$::=$	$\mathsf{unit} \mid \mathsf{bool} \mid \mathsf{int}$
Base types:	B	$::=$	$A \mid \mathsf{int\ ref}$
Typing environments:	Γ	$::=$	$\emptyset \mid x : B, \Gamma$

$$\frac{}{\Gamma \vdash v :\ \text{type of } v} \qquad \frac{x : B \in \Gamma}{\Gamma \vdash x : B} \qquad \frac{\Gamma \vdash x : \mathsf{int\ ref}}{\Gamma \vdash !x : \mathsf{int}}$$

$$\frac{\Gamma \vdash x : \mathsf{int\ ref} \qquad \Gamma \vdash e : \mathsf{int}}{\Gamma \vdash x := e : \mathsf{unit}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{unit} \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1 ;\ e_2 : A}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{bool} \qquad \Gamma \vdash e_2 : A \qquad \Gamma \vdash e_3 : A}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : A} \qquad \frac{\Gamma \vdash b : \mathsf{bool} \qquad \Gamma \vdash e : \mathsf{unit}}{\Gamma \vdash \mathsf{while}_\phi\ b\ \mathsf{do}\ e\ \mathsf{done} : \mathsf{unit}}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{int} \qquad \Gamma, x : \mathsf{int\ ref} \vdash e : A}{\Gamma \vdash \mathsf{let}\ x = \mathsf{ref}\ e_1\ \mathsf{in}\ e : A}$$

$$\frac{op : A_1 \to \cdots \to A_n \to A \qquad \Gamma \vdash e_i : A_i \quad i = 1, \ldots, n}{\Gamma \vdash (op\ e_1 \ldots e_n) : A}$$

$$\frac{\Gamma, x_1 : B_1, \ldots, x_n : B_n \vdash e : A \qquad \Gamma \vdash e_i : B_i \quad i = 1, \ldots, n}{\Gamma \vdash ([x_1 : B_1] \ldots [x_n : B_n] e\ e_1 \ldots e_n) : A}$$

Figure 5: Typing rules for Real

$$\frac{}{\{Q(X,X,v)\}\ v\ \{Q(X,X',\mathsf{r})\}} \quad \text{(value)}$$

$$\frac{}{\{Q(X,X,x)\}\ x\ \{Q(X,X',\mathsf{r})\}} \quad \text{(variable)}$$

$$\frac{}{\{Q(X,X,x)\}\ !x\ \{Q(X,X',\mathsf{r})\}} \quad \text{(access)}$$

$$\frac{\{P(X)\}\ e\ \{Q(X,X'[x' \leftarrow \mathsf{r}],\mathsf{void}\}}{\{P(X)\}\ x := e\ \{Q(X,X',\mathsf{r})\}} \quad \text{(store)}$$

$$\frac{\{P(X)\}\ e_1\ \{R(X,X',\mathsf{r})\} \qquad \{R(X_i,X,\mathsf{void}\}\ e_2\ \{Q(X_i,X',\mathsf{r})\}}{\{P(X)\}\ e_1; e_2\ \{Q(X,X',\mathsf{r})\}} \quad \text{(sequence)}$$

$$\frac{\{P(X)\}\ e_1\ \{R(X,X',\mathsf{r})\} \qquad \begin{array}{c}\{R(X_i,X,\mathsf{true}\}\ e_2\ \{Q(X_i,X',\mathsf{r})\}\\ \{R(X_i,X,\mathsf{false}\}\ e_3\ \{Q(X_i,X',\mathsf{r})\}\end{array}}{\{P(X)\}\ \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3\ \{Q(X,X',\mathsf{r})\}} \quad \text{(if)}$$

$$\frac{\{Q(X_i,X)\}\ b\ \{I(X_i,X',\mathsf{r})\} \qquad \{I(X_i,X,\mathsf{true})\}\ e\ \{Q(X_i,X') \wedge \phi[X \leftarrow X'] < \phi\}}{\{Q(X,X)\}\ \mathsf{while}_\phi\ b\ \mathsf{do}\ e\ \mathsf{done}\ \{Q(X,X') \wedge R(X,X',\mathsf{false})\}} \quad \text{(loop)}$$

$$\text{where } I(X,X',\mathsf{r}) \equiv Q(X,X') \wedge R(X,X',\mathsf{r})$$

$$\frac{\{P(X)\}\ e_1\ \{R(X,X',\mathsf{r})\} \qquad \{R(X_i,X,x\}\ e_2\ \{Q(X_i,X',\mathsf{r})\}}{\{P(X)\}\ \mathsf{let}\ x = \mathsf{ref}\ e_1\ \mathsf{in}\ e_2\ \{Q(X,X',\mathsf{r})\}} \quad \text{(new ref)}$$

Primitive operation

$$\frac{\{P(X)\}\ e_1\ \{R_1(X,X',\mathsf{r})\} \quad \{R_{i-1}(X_0,X,v_{i-1}\}\ e_i\ \{R_i(X_0,X',\mathsf{r})\} \quad i=2,\ldots,n}{\{P(X)\}\ (op\ e_1 \ldots e_n)\ \{Q(X,X',\mathsf{r})\}} \quad \text{(op)}$$

$$\text{where } R_n(X,X',\mathsf{r}) \equiv Q(X,X',(op\ v_1 \ldots v_{n-1}\ \mathsf{r}))$$

Function

$$\frac{\begin{array}{c}\{P(X)\}\ e_1\ \{R_1(X,X',\mathsf{r})\} \quad \{R_{i-1}(X_0,X,x_{i-1})\}\ e_i\ \{R_i(X_0,X',\mathsf{r})\} \quad i=2,\ldots,k\\ \{R_k(X_0,X,x_k)[y_i \leftarrow z_i]\}\ e\ \{Q(X_0,X',\mathsf{r})[y_i \leftarrow z_i]\}\end{array}}{\{P(X)\}\ (f\ e_1\ \ldots\ e_k\ y_1\ \ldots\ y_n)\ \{Q(X,X',\mathsf{r})\}} \quad (\beta)$$

$$\text{where } f \equiv [x_1 : A_1] \ldots [x_k : A_k][z_1 : \mathsf{int\ ref}] \ldots [z_n : \mathsf{int\ ref}]e$$

$$\frac{\begin{array}{c}\{P(X)\}\ e_1\ \{R_1(X,X',\mathsf{r})\} \quad \{R_{i-1}(X_0,X,x_{i-1})\}\ e_i\ \{R_i(X_0,X',\mathsf{r})\} \quad i=2,\ldots,k\\ R_k(X_0,X,x_k) \Rightarrow P_f(X)[z_i \leftarrow y_i]\\ R_k(X_0,X,x_k) \wedge Q_f(X,X',\mathsf{r})[z_i \leftarrow y_i] \Rightarrow Q(X_0,Y,W',\mathsf{r}) \quad Y \cap \mathsf{var}(f) = \emptyset\end{array}}{\{P(X)\}\ (f\ e_1\ \ldots\ e_k\ y_1\ \ldots\ y_n)\ \{Q(X,Y',W',\mathsf{r})\}} \quad \text{(app)}$$

Figure 6: Deduction rules for Real