A Hoare Logic for Call-by-Value Functional Programs

Yann Régis-Gianas[1] and François Pottier[2]

[1] INRIA Saclay - Île-de-France, ProVal, Orsay, F-91893
LRI, Université Paris-Sud, CNRS, Orsay, F-91405
[2] INRIA Paris - Rocquencourt, Gallium - Domaine de Voluceau - F-78153

Abstract. We present a Hoare logic for a call-by-value programming language equipped with recursive, higher-order functions, algebraic data types, and a polymorphic type system in the style of Hindley and Milner. It is the theoretical basis for a tool that extracts proof obligations out of programs annotated with logical assertions. These proof obligations, expressed in a typed, higher-order logic, are discharged using off-the-shelf automated or interactive theorem provers. Although the technical apparatus that we exploit is by now standard, its application to call-by-value functional programming languages appears to be new, and (we claim) deserves attention. As a sample application, we check the partial correctness of a balanced binary search tree implementation.

1 Introduction

Hoare logic [1, 2, 3] is a discipline for annotating programs with logical formulae, known as assertions, and for extracting logical formulae, known as proof obligations, out of such annotated programs. The validity of the proof obligations, which can be verified either manually or mechanically, entails the correctness of the annotated program. That is, it guarantees that the assertions are correct static predictions of the program's dynamic behavior.

Hoare logic was originally designed for a "while language", that is, a simple imperative programming language, equipped with an iteration construct and a fixed number of global, mutable variables. Recursive, higher-order procedures were the subject of much attention in the late 1970's and early 1980's [4, 5, 6, 7, 8]. More recently, heap-allocated, mutable data structures, as well as object-oriented features, have been deeply investigated. This has led to the development of practical specification languages and tools targeting, for instance, Java [9, 10, 11], C [12] and C# [13].

We would like to put forth the thesis that this traditional focus on imperative programming languages has been, to some extent, detrimental: it has consumed a great amount of energy, while comparatively little effort was being devoted to the key features that will be required in order for the methodology to scale up, such as modularity and abstraction. We would also like to raise a question: since functional programs are significantly easier to check for correctness, why hasn't this activity become routine in the functional programming community, forty years after Floyd and Hoare's seminal papers?

On the cost of imperative programming. There are several reasons why functional programming can be considered superior to imperative programming [14]. One of them is that functional programs are easier to reason about. In other words, there is a cost to reasoning about state.

In a typical modern imperative programming language, all heap-allocated data is mutable. As a result, instead of reasoning in terms of high-level entities such as, say, pairs, lists, trees, etc., programmers are forced to reason in terms of a view of the heap as a graph. More concretely, they must write down and prove formulae that involve mappings of memory addresses to memory blocks [12, 15].

The possibility of aliasing means that, whenever some memory block is written, the memory that is accessible through every type-compatible pointer is potentially affected. This makes it difficult to reason about the effects of a single write operation, and creates the problem of representation exposure [16, 17]. In order to address this issue, researchers have developed linear types and regions [18], ownership types [19], and separation logic [20], among other approaches.

Our research agenda. We do not claim that the above issues are not worth investigating: on the contrary, they are quite fascinating. However, it is a pity that we do not, today, have mature tools for checking the correctness of functional programs. This explains why, in this paper, we study a Hoare logic for (call-by-value) functional programs without state.

The programs that we are interested in checking rely heavily on (possibly higher-order) functions, algebraic data structures, and type polymorphism. We claim that it is quite easy to extract succinct and natural proof obligations out of such programs, provided, of course, that they are annotated with specifications.

There are two benefits to be reaped by not reasoning about state. As far as the user is concerned, this leads to simpler specifications and proof obligations. As far as the implementor is concerned, this saves a large part of the "implementation budget", which can then be spent on features such as type polymorphism, type abstraction, and modularity. The importance of these features cannot be overstated: in the end, the key to success is the ability to develop and check program components independently.

Contribution. In this paper, we present the design of a typed, polymorphic, higher-order programming language, where programs can be annotated with assertions expressed in a typed, polymorphic, higher-order logic. We define a procedure for extracting proof obligations out of programs, and show that it is sound. A publicly available prototype tool [21] has been developed, which works in conjunction with the interactive theorem prover Coq [22], with the automated first-order theorem prover Alt-Ergo [23], or with both at once. This tool has been used to check the partial correctness of several non-trivial data structure implementations, including balanced binary search trees and purely functional double-ended queues [24]. We hope to publish detailed accounts of these implementations in the future.

Highlights of our approach. Here are some of the key technical features of our approach.

We focus on partial correctness. We do not require programs to terminate, and do not generate proof obligations to ensure termination. It is up to the user to determine which properties of the code are of sufficient interest to deserve proof, and to insert assertions where desired. At one extreme, a program that contains no assertions leads to no proof obligations. There is no cost to be paid up front for using our methodology.

Our preconditions are prescriptive: it is impossible to call a function unless its precondition F_1 holds. A descriptive interpretation of preconditions can be simulated by using the precondition **true** and the postcondition $F_1 \Rightarrow F_2$. This allows unconditional invocation, and states that the function's result must satisfy F_2 if its argument satisfies F_1.

Values, programs, types, and logical formulae are distinct syntactic categories. Proofs do not necessarily appear within programs: proof obligations are delegated to an external theorem prover, which may or may not require or produce explicit proof terms.

We do not embed values, programs, or formulae within types. Thus, our types are first-order terms: they include type variables, parameterized algebraic data types, and function types, just as in ML. As a result, type inference in the style of Milner [25] is possible, and implemented in our tool [21]. Type inference does not generate any proof obligations. We do not have dependent types, such as lists indexed with an integer length [26], but simulate them as follows. Instead of declaring that x has type *list* n, we declare that x has type *list*, and assert the logical formula $\mathit{length}(x) = n$, where the function *length* is inductively defined at the logical level.

Formulae can refer to values, but not to expressions. This is important, because values are pure, whereas expressions are potentially impure. Although our logic cannot explicitly reason about state, it is nevertheless soundly applicable to programs that involve non-termination, non-determinism, input/output, or mutable state. (Reading an input stream, or dereferencing a pointer to mutable storage, can be viewed as non-deterministic operations.) In that case, it allows establishing properties that do not depend on the behavior of any impure operation. This means, for instance, that we can prove the partial correctness of a functional program even if it has been instrumented with possibly impure debugging, profiling, or logging instructions.

In our programming language, functions, which are potentially impure, are values, so they can appear within formulae. But what does it mean for a formula to refer to a computational function f of type, say, $\tau_1 \overset{\cdot}{\longrightarrow} \tau_2$? Our answer is to view f, at the logical level, as a pair of predicates, which represent f's precondition and postcondition. In other words, when used within a formula, f has type (roughly) $(\tau_1 \rightarrow \mathbf{prop}) \times (\tau_1 \rightarrow \tau_2 \rightarrow \mathbf{prop})$. The two pair projections, written **pre** and **post**, can be used to refer to the pair components. That is, $\mathbf{pre}(f)$ and $\mathbf{post}(f)$ offer lightweight notations for referring to f's precondition and postcondition. When f is a known (**let**-bound) function, this mechanism can be viewed merely as offering abbreviations for known formulae. However, when f is unknown (λ- or $\forall$-bound), it becomes key to writing natural specifications for higher-order functions (§7.5).

In summary, although the technical apparatus that we exploit is by now standard, we believe that it is worth drawing attention to the combination of power and simplicity offered by our technical choices. If extended with a suitable module system, and equipped with a compilation path down to, say, Objective Caml [27], our tool could be used to construct correct purely functional program components, possibly for use within larger, partly imperative programs.

Outline of the paper. The paper is laid out as follows. First, we briefly introduce a higher-order logic, in which assertions and proof obligations are expressed (§2). Then, we present the syntax and call-by-value semantics of a core functional programming language whose expressions carry explicit assertions (§3). We describe the type system, as well as the procedure for extracting proof obligations out of programs (§4). We present a few extensions of the language (§5) and discuss how proof obligations are transformed for submission to external theorem provers (§6). Last, we present a few excerpts of our balanced binary search tree implementation (§7) and review related work (§8).

2 The Underlying Logic

2.1 Syntax

We rely on a mostly standard higher-order logic [28] whose types and terms appear in Figure 1. Types θ include type variables α, parameterized inductive types, function types, product types, and the type **prop** of logical propositions. In the following, the syntax of terms is extended with standard syntactic sugar for falsity, disjunction, implication, equivalence, existential quantification, etc.

The typing rules appear in Figure 2. In general, we write t for terms of arbitrary type. We write F for *formulae*, that is, terms of type **prop**, and P for *predicates*, that is, terms of type $\theta \rightarrow \textbf{prop}$. The binary operator $\#$, used in several definitions, expresses the fact that two objects have no common free names.

Our logic is not simply-typed. Because our computational language (§3) is polymorphic, and because we wish to lift every computational value up to the logical level, we need polymorphism at the logical level as well. For this reason, we have logical type schemes $\varsigma ::= \forall\bar{\alpha}.\,\theta$, where $\bar{\alpha}$ is a vector of distinct type variables. Every occurrence of a variable x is explicitly applied to a type vector $\bar{\theta}$, which states how the type scheme associated with x is instantiated. For this reason also, we introduce universal quantification over type variables, and use *facts* of the form $\forall\bar{\alpha}.F$. Facts are not formulae: they do not appear in Figure 1. Facts appear only within computational-level type environments Γ (§3.1, Figure 3). The extension of higher-order logic with this very simple form of explicit quantification over types is embedded within the Calculus of Inductive Constructions (§2.2).

Logical Types

$$\begin{array}{lll}
\theta & ::= \alpha & \textit{Variable} \\
 & \mid d\,\bar{\theta} & \textit{Data} \\
 & \mid \theta \to \theta & \textit{Function} \\
 & \mid \theta \times \theta & \textit{Product} \\
 & \mid \mathbf{prop} & \textit{Proposition} \\
\varsigma & ::= \forall\bar{\alpha}.\,\theta & \textit{Scheme}
\end{array}$$

Logical Type Environments

$$\begin{array}{lll}
\Delta & ::= \emptyset & \textit{Nil} \\
 & \mid \Delta, (x : \varsigma) & \textit{Variable} \\
 & \mid \Delta, \bar{\alpha} & \textit{Type Variables}
\end{array}$$

Logical Terms

$$\begin{array}{lll}
t, F, P & ::= x\,\bar{\theta} & \textit{Variable} \\
 & \mid D\,\bar{\theta}\,(t, \ldots, t) & \textit{Data} \\
 & \mid \lambda(x : \theta).t & \textit{Abstraction} \\
 & \mid t(t) & \textit{Application} \\
 & \mid (t, t) & \textit{Product} \\
 & \mid \pi_1 & \textit{Projection (also written } \mathbf{pre}\textit{)} \\
 & \mid \pi_2 & \textit{Projection (also written } \mathbf{post}\textit{)} \\
 & \mid \mathbf{true} & \textit{Truth} \\
 & \mid t = t & \textit{Equality} \\
 & \mid t \wedge t & \textit{Conjunction} \\
 & \mid \neg\, t & \textit{Negation} \\
 & \mid \forall(x : \theta).t & \textit{Universal Quantification}
\end{array}$$

Fig. 1. The logic (syntax)

The logic offers parameterized inductive types. We assume that each inductive type constructor d carries a fixed integer arity, and that every application $d\,\bar{\theta}$ is arity-consistent. We further assume that d comes with a finite number of data constructors D, each of which is assigned a type scheme of the form:

$$\forall\bar{\alpha}.\,\theta_1 \times \ldots \times \theta_n \to d\,\bar{\alpha}$$

We impose a positivity condition [29], which is informally summed up as follows: in the above type scheme, the type constructor d (or any type constructor whose definition is mutually recursive with the definition of d) must not appear under the left-hand side of an arrow within $\theta_1, \ldots, \theta_n$.

Although there is an introduction form for inductive types, namely the application of a data constructor D, no elimination form is provided here. We can get away with this omission because the process of *extracting* proof obligations, which is the focus of the present paper, requires no such forms. Of course, when it comes to *discharging* proof obligations, that is, proving theorems, then inductive definitions and proofs become necessary.

$$\begin{aligned}(\Delta, (x : \varsigma))(x) &= \varsigma \\ (\Delta, (x_1 : \varsigma))(x_2) &= \Delta(x_2) \text{ if } x_1 \# x_2 \\ (\Delta, \bar{\alpha})(x) &= \Delta(x) \quad \text{if } \bar{\alpha} \# \Delta(x)\end{aligned}$$

$$\frac{\Delta(x) = \forall \bar{\alpha}.\, \theta}{\Delta \vdash x\,\bar{\theta} : [\bar{\alpha} \mapsto \bar{\theta}]\theta} \qquad \frac{\begin{array}{c} D : \forall \bar{\alpha}.\, \theta_1 \times \ldots \times \theta_n \to d\,\bar{\alpha} \\ \forall i \qquad \Delta \vdash t_i : [\bar{\alpha} \mapsto \bar{\theta}]\theta_i \end{array}}{\Delta \vdash D\,\bar{\theta}\,(t_1, \ldots, t_n) : d\,\bar{\theta}} \qquad \frac{\Delta, (x : \theta_1) \vdash t : \theta_2}{\Delta \vdash \lambda(x : \theta_1).t : \theta_1 \to \theta_2}$$

$$\frac{\begin{array}{c} \Delta \vdash t_1 : \theta_1 \to \theta_2 \\ \Delta \vdash t_2 : \theta_1 \end{array}}{\Delta \vdash t_1(t_2) : \theta_2} \qquad \frac{\forall i \qquad \Delta \vdash t_i : \theta_i}{\Delta \vdash (t_1, t_2) : \theta_1 \times \theta_2} \qquad \frac{\Delta \vdash t : \theta_1 \times \theta_2}{\Delta \vdash \pi_i(t) : \theta_i} \qquad \frac{}{\Delta \vdash \mathbf{true} : \mathbf{prop}}$$

$$\frac{\forall i \qquad \Delta \vdash t_i : \theta}{\Delta \vdash t_1 = t_2 : \mathbf{prop}} \qquad \frac{\forall i \qquad \Delta \vdash t_i : \mathbf{prop}}{\Delta \vdash t_1 \wedge t_2 : \mathbf{prop}} \qquad \frac{\Delta \vdash t : \mathbf{prop}}{\Delta \vdash \neg\, t : \mathbf{prop}}$$

$$\frac{\Delta, (x : \theta) \vdash t : \mathbf{prop}}{\Delta \vdash \forall (x : \theta).t : \mathbf{prop}} \qquad \frac{\Delta, \bar{\alpha} \vdash t : \mathbf{prop}}{\Delta \vdash \forall \bar{\alpha}.t : \mathbf{prop}}$$

Fig. 2. The logic (type system)

2.2 Interpretation

Our higher-order logic is embedded within the Calculus of Inductive Constructions [29, 30], abbreviated to CiC in the sequel. Indeed, each type of our logic can be translated into a term of CiC whose type is Type_0. This guarantees that the translation of polymorphic quantification only introduces type variables of type Type_0 in CiC. Each construct of our logic is directly mapped to its counterpart in CiC. This interpretation guarantees that our logic is consistent and validates a number of laws that are used in establishing the soundness of our system (§4.8).

3 The Computational Language

3.1 Syntax

The syntax of our programming language appears in Figure 3. It is equipped with an ML-style type system [25], so types τ and type schemes σ are distinguished. Types include type variables, parameterized algebraic data types, and function types. We write $\dot{\longrightarrow}$ for the computational function type constructor, so as to distinguish it from the logical function type constructor, written $\to$ (Figure 1).

We impose a syntactic separation between values and expressions, and require both operands of the function application operator, as well as **case** scrutinees, to be values. This imposes a style, reminiscent of A-normal form [31], where the result of every intermediate computation is named via a **let** construct. Of course, such a style is quite user-unfriendly, so, in practice, we offer an unrestricted

		Computational Types
τ	$::= \alpha$	*Variable*
	$\mid d\,\bar{\tau}$	*Data*
	$\mid \tau \overset{\cdot}{\longrightarrow} \tau$	*Function*
σ	$::= \forall\bar{\alpha}.\,\tau$	*Scheme*
		Computational Type Environments
Γ	$::= \emptyset$	*Nil*
	$\mid \Gamma, (x : \sigma)$	*Variable*
	$\mid \Gamma, \bar{\alpha}$	*Type Variables*
	$\mid \Gamma, \forall\bar{\alpha}.F$	*Assumption*
		Values
v	$::= x\,\bar{\tau}$	*Variable*
	$\mid D\,\bar{\tau}\,(v, \ldots, v)$	*Data*
	$\mid \mathbf{fun}\, f(x : \tau/F) : (x : \tau/F) =\ e$	*Recursive Function*
		Patterns
p	$::= x\,\bar{\tau}$	*Variable*
	$\mid D\,\bar{\tau}\,(p, \ldots, p)$	*Data*
		Expressions
e	$::= v$	*Value*
	$\mid v(v)$	*Function Application*
	$\mid \mathbf{let}\,(x\,\bar{\alpha} : \tau/F) = e\ \mathbf{in}\ e$	*Local Binding*
	$\mid \mathbf{case}\, v\ \mathbf{of}\ c$	*Pattern Matching*
		Cases
c	$::= \emptyset$	*Nil*
	$\mid (p \mapsto e)\ []\ c$	*Cons*

Fig. 3. The computation language (syntax)

surface language, and automatically translate it down to the kernel language described here.

The language supports type inference in the style of Hindley and Milner. However, in this paper, we are not concerned with type inference, so we work with explicitly-typed programs. This is visible (i) in the syntax of values and patterns, where variables and data constructors are annotated with vectors of types that indicate how polymorphic type schemes are instantiated, (ii) at **fun** and **let** constructs, where bound variables are annotated with types, and (iii) at **let** constructs, where a vector of type variables $\bar{\alpha}$ can be explicitly bound.

A function definition takes the general form:

$$\mathbf{fun}\, f(x_1 : \tau_1/F_1) : (x_2 : \tau_2/F_2) =\ e$$

The symbol / should be read "where". Every function is recursive, so that f is bound within e. The formal parameter x_1 is bound within the precondition

F_1, within the postcondition F_2, and within e. The variable x_2, which stands for the result of the function, is bound within the postcondition F_2. We require every function to be annotated with an explicit precondition and postcondition (if missing, **true** is assumed).

A local variable definition takes the general form:

$$\mathbf{let}\,(x\,\bar{\alpha} : \tau/F) = e_1\ \mathbf{in}\ e_2$$

The local variable x is bound within F and within e_2. The type variables $\bar{\alpha}$ are bound within τ, F, and e_1. The proposition F serves as a postcondition for e_1. If it is missing, a default postcondition is assumed, whose definition is deferred to §3.3.

A case analysis takes the general form:

$$\mathbf{case}\,v\,\mathbf{of}\,c$$

Here, c is a possibly empty sequence of cases (i.e., branches). Each branch is of the form $(p \mapsto e)$, where the variables that appear in the pattern p are bound within e. Patterns must be linear, that is, a pattern cannot bind a variable twice.

3.2 Lifting Computational Entities to the Logical Level

In a Hoare logic, formulae refer to values. That is, if x is bound, at the computational level, by a **fun**, **let**, or **case** construct, then it is possible for a formula F, embedded in the code within the scope of x, to refer to x. This raises two questions: first, if x has computational type τ, what is its logical type, to be used when typechecking F? Second, if, for the purposes of evaluation, x is substituted with a computational value v, what is the corresponding logical value, to be used when interpreting F?

The problem of lifting types and values to the logical level is trivial in a first-order language. Indeed, the type algebra only contains basic types which are translated to type constants (**int** is mapped to **int**). Besides, computational values are essentially first-order terms, interpreted as data in the logic. Yet, in an higher-order language, functions are first-class values. What should be the logical reflection of their code ?

We answer these questions by lifting both computational types and computational values up to the logical level (Figure 4). That is, to each computational type τ, we associate a logical type $\lceil\tau\rceil$, and to each computational value v, we associate a logical term $\lceil v\rceil$, with the intended property that if v has computational type τ, then $\lceil v\rceil$ has logical type $\lceil\tau\rceil$. Patterns are lifted too. Because patterns form a subset of values, no extra definitions are needed.

As announced (§1), computational functions are reflected, at the logical level, as pairs of a precondition and postcondition. This is made explicit in the lifting of computational function types:

$$\lceil\tau_1 \overset{\cdot}{\longrightarrow} \tau_2\rceil = (\lceil\tau_1\rceil \to \mathbf{prop}) \times (\lceil\tau_1\rceil \to \lceil\tau_2\rceil \to \mathbf{prop})$$

Types

$$\begin{aligned}\lceil\alpha\rceil &= \alpha\\ \lceil d\,\bar\tau\rceil &= d\,\lceil\bar\tau\rceil\\ \lceil\tau_1 \stackrel{\cdot}{\longrightarrow} \tau_2\rceil &= (\lceil\tau_1\rceil \to \mathbf{prop}) \times (\lceil\tau_1\rceil \to \lceil\tau_2\rceil \to \mathbf{prop})\end{aligned}$$

Type schemes

$$\lceil\forall\bar\alpha.\,\tau\rceil = \forall\bar\alpha.\,\lceil\tau\rceil$$

Type environments

$$\begin{aligned}\lceil\emptyset\rceil &= \emptyset\\ \lceil\Gamma,(x:\sigma)\rceil &= \lceil\Gamma\rceil,(x:\lceil\sigma\rceil)\\ \lceil\Gamma,\bar\alpha\rceil &= \lceil\Gamma\rceil,\bar\alpha\\ \lceil\Gamma,\forall\bar\alpha.F\rceil &= \lceil\Gamma\rceil\end{aligned}$$

Values

$$\begin{aligned}\lceil x\,\bar\tau\rceil &= x\,\lceil\bar\tau\rceil\\ \lceil D\,\bar\tau\,(v_1,\ldots,v_n)\rceil &= D\,\lceil\bar\tau\rceil\,(\lceil v_1\rceil,\ldots,\lceil v_n\rceil)\\ \lceil\mathbf{fun}\,f(x_1:\tau_1/F_1):(x_2:\tau_2/F_2) = e\rceil &= (\lambda(x_1:\lceil\tau_1\rceil).F_1, \lambda(x_1:\lceil\tau_1\rceil).\lambda(x_2:\lceil\tau_2\rceil).F_2)\end{aligned}$$

Fig. 4. Lifting computational types and values to the logical level

The first component of the pair, which represents the function's precondition, is abstracted over the function's argument, while the second component, which represents the postcondition, is abstracted over both argument and result.

As a result of this definition, if f is bound, at the computational level, to a function of type $\tau_1 \stackrel{\cdot}{\longrightarrow} \tau_2$, then a formula embedded within the code, in the scope of f, views f as a pair of predicates, and can refer to $\mathbf{pre}(f)$ and $\mathbf{post}(f)$. (Recall that, as per Figure 1, **pre** and **post** are sugar for the projections π_1 and π_2.) Note that f does not denote a logical function. Within a formula, an application $f(t)$ does not make sense: it is ill-typed.

Values of computational function type (that is, λ-abstractions) are lifted up to the logical level in a way that is consistent with this definition. A function's precondition and postcondition alone determine how it is lifted: its code is ignored. (The conformance of a function's body to its declared pre- and postcondition is checked, of course, via a proof obligation: see rule FUN in Figure 6.) This reflects a philosophy in which the only way of reasoning about the behavior of a function value is via its specification: code never appears within formulae.

In order to lift algebraic data types, we lift every algebraic data type definition into an isomorphic inductive type definition. So, for every computational-level algebraic data type constructor d, there must be a logical-level inductive type constructor, also written d, of identical arity. For every computational-level data constructor

$$D : \forall\bar\alpha.\,\tau_1 \times \ldots \times \tau_n \to d\,\bar\alpha,$$

there must be a logical-level data constructor

$$D : \forall\bar\alpha.\,\lceil\tau_1\rceil \times \ldots \times \lceil\tau_n\rceil \to d\,\bar\alpha.$$

Due to the manner in which computational function types are lifted, the positivity condition (§2) requires the type constructor d to not appear *under any side* of a computational arrow within $\tau_1, \ldots, \tau_n$. This can be a limitation (§9).

3.3 Inferring Strongest Postconditions

In order to simplify the definition of the procedure that extracts proof obligations, we have required every **let** construct to carry an explicit postcondition for its left-hand sub-expression (§3.1). In practice, however, annotating every **let** construct would be quite unpleasant, so it is desirable to construct a reasonable postcondition when the user does not provide one.

Ideally, the formula that we should construct in such a situation is the *strongest postcondition* of the left-hand sub-expression. Our logic is, in fact, sufficiently powerful to express strongest postconditions for every construct in our programming language. For instance, the strongest postcondition for a value v is $\lambda x.(x = \lceil v \rceil)$. The strongest postcondition for a function application $v_1(v_2)$ is $\mathbf{post}(\lceil v_1 \rceil)(\lceil v_2 \rceil)$. We could go on and explain how to construct strongest postconditions for **let** and **case** constructs. However, in these two cases, they would be complex formulae, involving existential quantification and disjunction.

Eventually, the postconditions carried by **let** constructs become part of proof obligations, where they appear as hypotheses. For this reason, we do not want them to be too complex: we wish to produce simple, comprehensible proof obligations.

Our answer to this issue is to construct strongest postconditions for values and function applications, as suggested above, but not for **let** and **case** constructs: instead, we rely on the user-provided postcondition, if there is one, or use the trivial postcondition **true**, otherwise.

In practice, when is it necessary for the user to provide an explicit annotation? The left-hand side of a **let** construct can be one of four expression forms: a value, a function application, a **let** form, or a **case** form. In the first two cases, we do use a strongest postcondition. The third case can be made to never happen, up to a conversion to A-normal form [31]. Only the last case remains. In summary, the only case where our simple-minded approach may call for an explicit, user-provided annotation is that of a **let** construct whose left-hand sub-expression is a **case** construct.

3.4 Notions of Substitution

Neither types nor formulae influence execution, but do appear in the syntax of values and expressions, in order to allow stating subject reduction and proving the soundness of our Hoare logic. So, the operational semantics reduces expressions that contain explicit types and formulae. To ensure that these annotations remain consistent as expressions are transformed, we must define a few slightly non-standard notions of substitution.

A single type variable α can appear within logical types as well as within computational types. Similarly, a single variable x can appear within formulae as well as within expressions. For this reason, we write $[\alpha \mapsto \tau]$ for the substitution that replaces every free occurrence of α at the computational level with τ and every free occurrence of α at the logical level with $\lceil \tau \rceil$. Similarly, we write $[x \mapsto v]$ for the substitution that replaces every free occurrence of x at the computational level with v and every free occurrence of x at the logical level with $\lceil v \rceil$.

We have annotated **let** constructs with explicit type abstractions and occurrences of variables with explicit type applications. As a result, contracting a **let**-redex requires contracting type-level β-redexes as well. In order to do so, we write $[x \mapsto \Lambda\bar{\alpha}.v]$ for a substitution that replaces every variable occurrence of the form $x\,\bar{\tau}$ with $[\bar{\alpha} \mapsto \bar{\tau}]v$. Again, this replacement is performed at both computational and logical levels, up to a lifting operation in the latter case.

Last, the notation $[x \mapsto v]$, which denotes a substitution of a value for a variable, is extended to the notation $[p \mapsto v]$, which, when p does not match v, is undefined, and, when p does match v, denotes a simultaneous substitution of values for variables, as follows. The formal definition is:

$$[D\,\bar{\tau}\,(p_1, \ldots, p_n) \mapsto D\,\bar{\tau}\,(v_1, \ldots, v_n)]$$

stands for

$$[p_1 \mapsto v_1] \cup \ldots \cup [p_n \mapsto v_n]$$

Because patterns are linear, this is a union of substitutions whose domains are pairwise disjoint.

3.5 Operational Semantics

A standard small-step, call-by-value operational semantics appears in Figure 5. There are three kinds of redexes (β, **let**, and **case**) and one evaluation context (the left-hand side of a **let** construct). An expression is *stuck* if it is irreducible and not a value. It is easy to check that an expression is stuck if and only if it contains, within an evaluation context, a sub-expression of the form $v_1(v_2)$, where v_1 is not a syntactic function, or of the form $\mathbf{case}\,v\,\mathbf{of}\,\emptyset$.

$$\begin{array}{rl}
v_1(v_2) \rightarrow & [x \mapsto v_2][f \mapsto v_1]e \\
& \text{if } v_1 \text{ is } \mathbf{fun}\,f(x : \tau/F) : (\ldots) = e \\[1ex]
\mathbf{let}\,(x\,\bar{\alpha} : \tau/F) = v\ \mathbf{in}\ e \rightarrow & [x \mapsto \Lambda\bar{\alpha}.v]e \\[1ex]
\mathbf{case}\,v\,\mathbf{of}\,(p \mapsto e)\ []\ c \rightarrow & [p \mapsto v]e \\
& \text{if } [p \mapsto v] \text{ is defined} \\[1ex]
\mathbf{case}\,v\,\mathbf{of}\,(p \mapsto e)\ []\ c \rightarrow & \mathbf{case}\,v\,\mathbf{of}\,c \\
& \text{if } [p \mapsto v] \text{ is undefined} \\[1ex]
\mathbf{let}\,(x\,\bar{\alpha} : \tau/F) = e_1\ \mathbf{in}\ e_2 \rightarrow & \mathbf{let}\,(x\,\bar{\alpha} : \tau/F) = e_1'\ \mathbf{in}\ e_2 \\
& \text{if } e_1 \rightarrow e_1'
\end{array}$$

Fig. 5. Operational semantics

4 The Type System and Proof System

We now equip the computational language with an ML-style type system and with a proof system (a Hoare logic), which can be viewed as an algorithm for extracting proof obligations out of well-typed programs. For the sake of succinctness, both are described using a single set of judgements, which assert at once that a program is well-typed and is annotated with consistent formulae.

In practice, our tool [21] first checks that the program is well-typed, and, at the same time, infers any omitted type annotations. Then, a set of proof obligations, expressed in our typed higher-order logic, is extracted. The fact that the program (including embedded formulae) is well-typed guarantees that the proof obligations are in turn well-typed.

4.1 Environments

The syntax of type environments Γ appears in Figure 3. As is standard, type environments bind variables and type variables. Environments also contain assumptions, that is, formulae that become hypotheses when proof obligations are emitted. An environment of the form $\Gamma, \forall\bar{\alpha}.F$ is well-formed when $\forall\bar{\alpha}.F$ has type **prop** under $\lceil\Gamma\rceil$.

4.2 Proof Obligations

A proof obligation is a judgement of the form $\Gamma \models F$, where F has type **prop** under $\lceil\Gamma\rceil$. The semantics of the judgment is the validity of the interpretation of F in CiC under the interpretation of the environment Γ, which is decided via an external theorem prover.

4.3 Judgements

The proof system is defined via three judgements, which state properties about values, patterns, and expressions, respectively:

Values	$\Gamma \vdash v : \tau$	(Figure 6)
Patterns	$\Gamma \vdash p : \tau$	(Figure 7)
Expressions	$\Gamma \vdash e : \tau \ \{P\}$	(Figure 8)

4.4 Values

The judgement $\Gamma \vdash v : \tau$ (Figure 6) states that, under the type environment Γ, the value v has type τ. No precondition or postcondition appear in the judgement. Indeed, because values require no computation, they never have a precondition. Furthermore, because all values can be lifted up to the logical level, they don't need an explicit postcondition: the strongest possible postcondition of a value v is simply equality with $\lceil v\rceil$.

$$\begin{aligned}
(\Gamma, (x : \sigma))(x) &= \sigma \\
(\Gamma, (x_1 : \sigma))(x_2) &= \Gamma(x_2) \text{ if } x_1 \# x_2 \\
(\Gamma, \bar{\alpha})(x) &= \Gamma(x) \ \text{ if } \bar{\alpha} \# \Gamma(x) \\
(\Gamma, \forall \bar{\alpha}.F)(x) &= \Gamma(x)
\end{aligned}$$

$$\textsc{Var}\quad \frac{\Gamma(x) = \forall \bar{\alpha}.\, \tau}{\Gamma \vdash x\, \bar{\tau} : [\bar{\alpha} \mapsto \bar{\tau}]\tau}
\qquad
\textsc{Data}\quad \frac{D : \forall \bar{\alpha}.\, \tau_1 \times \ldots \times \tau_n \to d\, \bar{\alpha} \qquad \forall i \quad \Gamma \vdash v_i : [\bar{\alpha} \mapsto \bar{\tau}]\tau_i}{\Gamma \vdash D\, \bar{\tau}\, (v_1, \ldots, v_n) : d\, \bar{\tau}}$$

$$\textsc{Fun}\quad \frac{\begin{array}{c} f \# F_1, F_2 \\ \lceil \Gamma, (x_1 : \tau_1) \rceil \vdash F_1 : \mathbf{prop} \qquad \lceil \Gamma, (x_1 : \tau_1), (x_2 : \tau_2) \rceil \vdash F_2 : \mathbf{prop} \\ \Gamma, (f : \tau_1 \stackrel{\cdot}{\longrightarrow} \tau_2), f = \lceil \mathbf{fun}\, f \ldots \rceil, (x_1 : \tau_1), F_1 \vdash e : \tau_2 \ \{\lambda(x_2 : \lceil \tau_2 \rceil).F_2\} \end{array}}{\Gamma \vdash \mathbf{fun}\, f(x_1 : \tau_1/F_1) : (x_2 : \tau_2/F_2) = \ e : \tau_1 \stackrel{\cdot}{\longrightarrow} \tau_2}$$

Fig. 6. The computation language (proof system: values)

Rules Var and Data are straightforward. Rule Fun is more complex. Two premises require the precondition F_1 and postcondition F_2 to be well-formed formulae, under appropriate environments. The last premise checks that the function's body conforms to the function's specification. In order to do so, the type environment is extended with bindings for f and x_1. It is also extended with the hypothesis

$$f = \lceil \mathbf{fun}\, f \ldots \rceil,$$

which by definition of lifting (Figure 4) is synonymous for

$$f = (\lambda(x_1 : \lceil \tau_1 \rceil).F_1, \lambda(x_1 : \lceil \tau_1 \rceil).\lambda(x_2 : \lceil \tau_2 \rceil).F_2).$$

This hypothesis gives meaning to occurrences of $\mathbf{pre}(f)$ and $\mathbf{post}(f)$ within the body of the function, allowing recursive calls to f to be checked. Last, the environment is also extended with the precondition F_1, which means that, within the body of the function, the precondition is assumed to hold. Under this extended environment, the body of the function is required to produce a value that meets the postcondition $\lambda(x_2 : \lceil \tau_2 \rceil).F_2$.

It is not difficult to see that $\Gamma \vdash v : \tau$ implies $\lceil \Gamma \rceil \vdash \lceil v \rceil : \lceil \tau \rceil$. This property is required for the typing rules to construct only well-formed formulae.

4.5 Patterns

The judgement $\Gamma \vdash p : \tau$ (Figure 7) states that a value of type τ can safely be matched against the pattern p, giving rise to (exactly) the bindings described by Γ. As in ML, these bindings are monomorphic (see Pat-Var). Because patterns are linear, the type environments $\Gamma_1, \ldots, \Gamma_n$ in Pat-Data have disjoint domains.

$$\textsc{Pat-Var}\quad (x:\tau)\vdash x:\tau \qquad\qquad \textsc{Pat-Data}\quad \frac{D:\forall\bar{\alpha}.\,\tau_1\times\ldots\times\tau_n\to d\,\bar{\alpha} \qquad \forall i \quad \Gamma_i\vdash p_i:[\bar{\alpha}\mapsto\bar{\tau}]\tau_i}{\Gamma_1,\ldots,\Gamma_n\vdash D\,\bar{\tau}\,(p_1,\ldots,p_n):d\,\bar{\tau}}$$

Fig. 7. The computation language (proof system: patterns)

4.6 Expressions

The judgement $\Gamma\vdash e:\tau\ \{P\}$ (Figure 8) states that, under the type environment Γ, the expression e has type τ and (if it terminates) produces a value whose logical reflection satisfies the predicate P. In such a judgement, P has type $\lceil\tau\rceil\to\mathbf{prop}$ under $\lceil\Gamma\rceil$.

Rule Value directly reflects this intended meaning: the judgement $\Gamma\vdash v:\tau\ \{P\}$ holds if and only if v has type τ under Γ and its logical reflection $\lceil v\rceil$ provably satisfies P under the hypotheses found in Γ. The premise $\Gamma\models P(\lceil v\rceil)$ is a proof obligation.

Rule App requires the function v_1 and its actual argument v_2 to have matching computational types. Furthermore, it emits two proof obligations, stating that (i) the actual argument must satisfy the function's precondition, and (ii) the function's postcondition must imply the desired postcondition P. In the last premise, we write $P'\Rightarrow P$, where P' and P have type $\lceil\tau_2\rceil\to\mathbf{prop}$, for $\forall(x:\lceil\tau_2\rceil).(P'(x)\Rightarrow P(x))$, where x is fresh for P' and P.

Rule Let checks that e_1 has type τ_1 and that e_1 complies with the postcondition F. Then, the rule performs type generalization, in the style of Milner [25], so

$$\textsc{Value}\quad \frac{\Gamma\vdash v:\tau \qquad \Gamma\models P(\lceil v\rceil)}{\Gamma\vdash v:\tau\ \{P\}} \qquad\qquad \textsc{App}\quad \frac{\Gamma\vdash v_1:\tau_1\overset{\cdot}{\longrightarrow}\tau_2 \quad \Gamma\vdash v_2:\tau_1 \quad \Gamma\models\mathbf{pre}(\lceil v_1\rceil)(\lceil v_2\rceil) \quad \Gamma\models\mathbf{post}(\lceil v_1\rceil)(\lceil v_2\rceil)\Rightarrow P}{\Gamma\vdash v_1(v_2):\tau_2\ \{P\}}$$

$$\textsc{Let}\quad \frac{x\,\#\,P \qquad \lceil\Gamma,\bar{\alpha},(x:\tau_1)\rceil\vdash F:\mathbf{prop} \qquad \Gamma,\bar{\alpha}\vdash e_1:\tau_1\ \{\lambda(x:\lceil\tau_1\rceil).F\} \qquad \Gamma,(x:\forall\bar{\alpha}.\,\tau_1),\forall\bar{\alpha}.[x\mapsto x\,\bar{\alpha}]F\vdash e_2:\tau_2\ \{P\}}{\Gamma\vdash\mathbf{let}\,(x\,\bar{\alpha}:\tau_1/F)=e_1\ \mathbf{in}\ e_2:\tau_2\ \{P\}}$$

$$\textsc{Case-Nil}\quad \frac{\Gamma\vdash v:\tau \qquad \Gamma\models\mathbf{false}}{\Gamma\vdash\mathbf{case}\,v\,\mathbf{of}\,\emptyset:\tau'\ \{P\}}$$

$$\textsc{Case-Cons}\quad \frac{\Gamma\vdash v:\tau \qquad \Gamma'\vdash p:\tau \qquad p\,\#\,v,P \qquad \Gamma,\Gamma',\lceil v\rceil=\lceil p\rceil\vdash e:\tau'\ \{P\} \qquad \Gamma,(\forall\Gamma'.\lceil v\rceil\neq\lceil p\rceil)\vdash\mathbf{case}\,v\,\mathbf{of}\,c:\tau'\ \{P\}}{\Gamma\vdash\mathbf{case}\,v\,\mathbf{of}\,(p\mapsto e)\,[]\,c:\tau'\ \{P\}}$$

Fig. 8. The computation language (proof system: expressions)

that e_2 is checked under the assignment $(x : \forall\bar{\alpha}.\,\tau_1)$. The hypothesis F is changed into $\forall\bar{\alpha}.[x \mapsto x\,\bar{\alpha}]F$, so as to reflect the fact that x now has polymorphic type.

In the operational semantics, a **let** construct behaves just like a β-redex. This suggests that it could perhaps be treated as syntactic sugar, obviating the need for the Let rule. However, this is not possible, for two reasons. One is that **let** allows type generalization, as explained above, whereas a β-redex does not. The other is that an appropriate postcondition for the function $\lambda x.e_2$ cannot be determined prior to extracting proof obligations: indeed, it has to be $\lambda x.P$, where P is computed only at extraction time.

Rule Case-Nil emits the proof obligation $\Gamma \models$ **false**, which requires the conjunction of hypotheses found within Γ to be inconsistent. This ensures that a **case** construct with zero branches is never executed.

Rule Case-Cons requires the value v and the pattern p to have a common type τ. The environment Γ' collects the variables bound by p, together with their types. Under the hypothesis that a certain instance of p matches v, which is expressed by extending Γ with Γ' and with the hypothesis $\lceil v \rceil = \lceil p \rceil$, the branch e must have the desired type τ' and meet the desired postcondition P. Furthermore, under the hypothesis that no instance of p matches v, which is written $\forall\Gamma'.\lceil v \rceil \neq \lceil p \rceil$, the remaining branches must have type τ' and meet the postcondition P. (Our use of $\lceil p \rceil$ exploits the fact that patterns form a subset of values, a welcome but unessential property.)

When checking a **case** construct with n branches, the $(k+1)$-th branch is checked under the assumption that none of the patterns $p_1, \ldots, p_k$ match the value v. In particular, for $k = n$, the conjunction of all hypotheses of the form $(\forall\Gamma'_i.\lceil v \rceil \neq \lceil p_i \rceil)$ is required to be inconsistent. This ensures that control cannot fall off the end of a **case** construct, or, in other words, that the case analyses are exhaustive. Today's ML and Haskell compilers implement a sound approximation to this check, using a purely syntactic criterion. We also implement this syntactic criterion: when it succeeds, emitting a proof obligation is unnecessary.

4.7 Algorithmic Reading

The judgement $\Gamma \vdash e : \tau\ \{P\}$ defines an algorithm for generating proof obligations. All four parameters of the judgement, namely Γ, e, τ, and P, are inputs of the algorithm, which attempts to build a derivation of the judgement by starting at the root of the expression e and working its way down into the sub-expressions of e. As the algorithm descends, entering **fun**, **let**, and **case** constructs, the environment Γ grows, accumulating new bindings and assumptions. At the same time, the postcondition P is propagated down, in a very straightforward process. At **let** constructs, this propagation process relies on the (default or user-provided, see §3.3) annotation in order to determine which postcondition must be propagated into the left-hand sub-expression. The output of the algorithm consists of the proof obligations, of the form $\Gamma \models F$, carried by the leaves of the derivation (see Value, App, and Case-Nil).

4.8 Soundness

The soundness of our type system and proof system is established in a standard, syntactic manner. The proofs appear in the first author's dissertation [32]. It states that the types and logical assertions carried by a program are a sound approximation of its dynamic semantics.

Lemma 1 (Environment Weakening). *$\Gamma_1, F, \Gamma_2 \vdash e : \tau \ \{P\}$ and $\Gamma_1 \models F$ imply $\Gamma_1, \Gamma_2 \vdash e : \tau \ \{P\}$.*

Lemma 2 (Postcondition Weakening). *$\Gamma \vdash e : \tau \ \{P_1\}$ and $\Gamma \models P_1 \Rightarrow P_2$ imply $\Gamma \vdash e : \tau \ \{P_2\}$.*

Lemma 3 (Type Substitution). *Let ϕ stand for $[\bar{\alpha} \mapsto \bar{\tau}]$. Then, $\Gamma_1, \bar{\alpha}, \Gamma_2 \vdash e : \tau \ \{P\}$ and $\bar{\alpha} \# dom(\Gamma_2)$ imply*

$$\Gamma_1, \phi(\Gamma_2) \vdash \phi(e) : \phi(\tau_2) \ \{\phi(P)\}$$

Lemma 4 (Value Substitution). *Let ρ stand for $[x \mapsto \Lambda\bar{\alpha}.v]$. Then, $\Gamma_1, (x : \forall\bar{\alpha}.\, \tau_1), \Gamma_2 \vdash e : \tau_2 \ \{P\}$ and $\Gamma_1, \bar{\alpha} \vdash v : \tau_1$ and $x \notin dom(\Gamma_2)$ imply*

$$\Gamma_1, \rho(\Gamma_2) \vdash \rho(e) : \tau_2 \ \{\rho(P)\}$$

Lemma 5 (Pattern Matching). *Let $\emptyset \vdash v : \tau$ and $\Gamma' \vdash p : \tau$ and $p \# v$. Then, $[p \mapsto v]$ is defined if and only if the formula $\exists\Gamma'.\lceil v \rceil = \lceil p \rceil$ is valid.*

Theorem 6 (Subject Reduction). *$\Gamma \vdash e : \tau \ \{P\}$ and $e \rightarrow e'$ imply $\Gamma \vdash e' : \tau \ \{P\}$.*

Theorem 7 (Progress). *$\emptyset \vdash e : \tau \ \{P\}$ implies that e is either reducible or a value v such that $P(\lceil v \rceil)$ is valid.*

5 A Few Extensions

Extra assertions. The following construct allows inserting an assertion at an arbitrary point in the code:

$$\textbf{assert}\ F\ \textbf{in}\ e$$

This construct requires F to hold: a proof obligation is emitted. It has no computational content: dynamically, it behaves like e. It is syntactic sugar for **let** $(x : \text{unit}/F) = ()$ **in** e, where x is fresh. It is particularly useful when our tool is used in conjunction with an automated theorem prover: if the theorem prover fails to discharge a proof obligation, the user can use **assert** to cut the proof into smaller, easier steps (if the proof obligation is in fact valid) or to find out what is wrong with the specification (if the proof obligation is in fact invalid).

The construct **absurd**, which statically requires **false** to hold, marks a piece of code as inaccessible. It is syntactic sugar for a **case** construct with zero branches.

Ghost variables and ghost parameters. It is sometimes desirable to explicitly introduce a *ghost variable*, that is, a name for a witness to an existentially quantified hypothesis. For this purpose, we suggest writing

$$\textbf{let logic}\ x : \theta / F\ \textbf{in}\ e$$

This construct binds x within F and e. It requires the assertion $\exists(x : \theta).F$ to hold, and introduces F as a new hypothesis into the context. Assertions embedded within e can refer to x, and their proofs can exploit the hypothesis F. However, occurrences of x at the computational level within e are forbidden, since "**let logic**" has no computational content.

Similarly, it is sometimes desirable to abstract a function with respect to a ghost parameter x, like this:

$$\textbf{fun}\, f[x : \theta](x_1 : \tau_1/F_1) : (x_2 : \tau_2/F_2) = e$$

The brackets bind a ghost parameter x within F_1, F_2, and e. (Again, occurrences of x at the computational level within e are forbidden.) Note that θ can be an arbitrary logical type, so this extension allows explicitly abstracting a function with respect to a proposition or predicate, if desired (see §7.5). Ghost variables and ghost parameters can in principle be viewed as syntactic sugar and translated away [33]. In a realistic implementation, however, they should be primitive notions.

6 Interfacing with External Theorem Provers

The overall verification process, implemented in our prototype tool, is composed of three main steps. First, type inference translates an implicitly typed source code into an explicitly typed internal language, very similar to the language formalized in §3. Second, the rules of the proof system defined in §4 are applied, producing a set of proof obligations. Third, these proof obligations are turned into goals of the two external provers Coq [22] and Alt-Ergo [23]. We describe this last step in the following.

6.1 Coq

Our typed, higher-order logic is easily embedded within the Calculus of Inductive Constructions, which underlies Coq. As a result, exporting proof obligations to Coq is a simple matter of pretty-printing. Implicit type instantiations are handled by Coq's system of implicit arguments. We could have made type instantiations explicit but this would have worsened readability.

Coq is an interactive theorem prover. In order to discharge a proof obligation, the user writes a proof script. An open problem is how to maintain these scripts as the source code of the program evolves. The location in the code where a proof obligation arises might change. The statement of a proof obligation might change as well. Perhaps a solution would be to allow only explicitly-stated, explicitly-named, lemmas to be proved interactively, and to rely solely on an automated theorem prover for discharging anonymous proof obligations, possibly by appeal to an explicit lemma.

6.2 Alt-Ergo

Alt-Ergo [23] is a fully automated theorem prover for a typed, polymorphic, first-order logic. Its design is partly inspired by Simplify [34]. However, Alt-Ergo's logic is typed and polymorphic, whereas Simplify's is untyped. This makes Alt-Ergo superior, from our point of view, to Simplify. Indeed, provided our proof obligations lie in the first-order fragment of our logic, they can be directly exported towards Alt-Ergo. If, on the other hand, we wished to use Simplify, we would have to encode our typed, polymorphic logic into Simplify's untyped logic. Such encodings have been studied [35], but are complex and costly. Of course, the trivial encoding that erases all types is unsound.

In addition to first-order logic, Alt-Ergo has native support for linear arithmetic and for the theory of constructors (that is, function symbols f such that $f(x) = f(y)$ implies $x = y$). The latter is useful for reasoning efficiently about algebraic data structures.

In the general case, our proof obligations are most naturally expressed in a higher-order logic, as shown in this paper. However, higher-order logic can be encoded into first-order logic. A standard encoding introduces "apply" predicates that help simulate β-conversion [36].

Perhaps surprisingly, in our case, this encoding can be made to look fairly natural. The symbols **pre** and **post**, which so far have stood for the pair projections, can be turned into predicates and simulate not only projection, but also application. Furthermore, we can make **pre** a binary predicate and **post** a ternary predicate, avoiding curried function applications. That is, instead of the higher-order formula:

$$f = (\lambda(x_1 : \lceil\tau_1\rceil).F_1, \lambda(x_1 : \lceil\tau_1\rceil).\lambda(x_2 : \lceil\tau_2\rceil).F_2),$$

we can write:

$$\begin{array}{r}\forall(x_1 : \lceil\tau_1\rceil).(\mathbf{pre}(f, x_1) \Leftrightarrow F_1)\\ \wedge\ \forall(x_1 : \lceil\tau_1\rceil).\forall(x_2 : \lceil\tau_2\rceil).(\mathbf{post}(f, x_1, x_2) \Leftrightarrow F_2)\end{array}$$

The pair and the three λ-abstractions have been η-expanded, and the projection and application symbols have been fused into applications of **pre** and **post**. Provided F_1 and F_2 are first-order formulae, this is a first-order formula.

Under this encoding, the definition of the lifting operation on computational types is modified so that the computational function type constructor is no longer interpreted:

$$\lceil\tau_1 \stackrel{\cdot}{\longrightarrow} \tau_2\rceil = \lceil\tau_1\rceil \stackrel{\cdot}{\longrightarrow} \lceil\tau_2\rceil$$

That is, we make $\stackrel{\cdot}{\longrightarrow}$ an uninterpreted binary type constructor at the logical level, so that the lifting of types becomes the identity. Thus, in the above formula, f has logical type $\tau_1 \stackrel{\cdot}{\longrightarrow} \tau_2$. The type schemes assigned to **pre** and **post** are as follows:

$$\begin{array}{c}\mathbf{pre} : \forall\alpha_1\alpha_2.\,(\alpha_1 \stackrel{\cdot}{\longrightarrow} \alpha_2) \times \alpha_1 \rightarrow \mathbf{prop}\\ \mathbf{post} : \forall\alpha_1\alpha_2.\,(\alpha_1 \stackrel{\cdot}{\longrightarrow} \alpha_2) \times \alpha_1 \times \alpha_2 \rightarrow \mathbf{prop}\end{array}$$

These declarations are admissible by Alt-Ergo. We believe that it should be possible to go a long way with first-order logic alone, even when the program exploits higher-order functions. However, at present, more practical experience is needed in order to support this conjecture.

7 Application: Finite Sets as Binary Search Trees

As an initial benchmark for our tool [21], we have transcribed Objective Caml's library implementation of finite sets, represented as balanced binary search trees, into our programming language. The code is presented in the concrete syntax of our prototype implementation.

7.1 Parameters

In the following, we fix a type "elt" of elements. We assume that an algebraic data type "bool", whose data constructors are "true" and "false", is available. We assume that an equality check over elements, written "=", is given. It is a function of computational type elt $\times$ elt $\dot{\longrightarrow}$ bool, whose specification could be written as follows:

$$\mathbf{post}(=, x_1, x_2, b) \Leftrightarrow (b = \text{true} \Leftrightarrow x_1 = x_2)$$

Similarly, we assume that an ordering relation, written "<", of logical type elt $\rightarrow$ elt $\rightarrow$ **prop**, is given, together with an ordering check, also written "<", of computational type elt $\times$ elt $\dot{\longrightarrow}$ bool, such that the latter decides the former.

We assume that a type of sets of elements, written "set", is available at the logical level, together with the standard operations (empty set, singleton set, union, membership, etc.) and a number of axioms or theorems that describe the properties of these operations.

In a full-scale programming language, our balanced binary search tree implementation would be a functor, parameterized over the types "elt" and "set", as well as as their operations and axioms.

7.2 Definitions

Figure 9 contains the definition of the algebraic data type "tree", of the logical-level inductive function "elements", and of the inductive predicate "bst". (The concrete syntax is provisional.) A binary tree is either empty or a binary node, carrying a root element, left and right sub-trees, and a cached measure of the tree's height. Our binary search trees are intended to implement a finite set abstraction. The logical function "elements" maps a binary tree to the finite set that it represents. It is defined by induction over the algebraic data type "tree". The property of being a binary search tree is defined by the inductive predicate "bst".

In the definition of "bst", the types of the universally quantified variables "x", "l", "r", "h", "y" are inferred. The types of the function "elements" and of the

```
type tree =
| Empty : tree
| Node : (int × tree × elt × tree) → tree

fixpoint elements : tree → set =
| Empty → empty
| Node (h, l, x, r) → elements (l) ∪ singleton (x) ∪ elements (r)

inductive bst : tree → prop =
| bst (Empty)
| ∀ (h, l, x, r).
    bst (l) and bst (r) and sup (x, elements (l)) and inf (x, elements (r))
    ⇒ bst (Node (h, l, x, r))
```

Fig. 9. Definitions for binary search trees

predicate “bst” could also be inferred, if desired. In practice, type annotations can always be omitted, except where polymorphic recursion is required.

The definition of “bst” constrains neither the shape of the tree nor the cached height information. This is done by another inductive predicate, named “avl” (not shown). In contrast with the “dependent types” [26, 37, 38] and “generalized algebraic data types” [39] schools, we favor a programming style in which invariants are not necessarily hardwired into data structures at definition time.

7.3 Membership in a Binary Search Tree

Figure 10 shows a function, “member”, that checks whether an element “x” is a member of a tree “t”. The precondition “bst(t)” requires “t” to be a binary search tree, but does not require it to be balanced, since this is not necessary for correctness. If one wished to (informally) ensure a logarithmic complexity bound, one could strengthen the precondition by adding the requirement “avl(t)”. This illustrates how a single data structure can be equipped with multiple invariants, not all of which are necessarily enforced at all times. The postcondition states that the Boolean result tells whether “x” is a member of the set implemented

```
let rec mem_bst (t, x) where bst (t)
returns b where ((b = true) ⇔ (x ∈ elements (t)))
= match t with
  | Empty → false
  | Node (h, l, y, r) →
    if (x = y) then true
    else if (x < y) then mem_bst (l, x)
    else mem_bst (r, x)
  end
```

Fig. 10. Membership in a binary search tree

by the tree “t”. No type annotations are needed in this definition. All types are inferred.

7.4 First-Order Iteration

We now define and specify first-order, persistent iterators [40] over binary search trees. Their expressive power surpasses that of “fold” (§7.5), yet their specification is simpler.

The implementation appears in Figure 11. An iterator is represented as a list of trees, which can be thought of as a stack in a depth-first traversal of some larger tree. (The definition of the type “list”, whose constructors are “Nil” and “Cons”, is omitted.)

To an iterator “i”, there corresponds a set of elements, which we write “remaining(i)”. Its inductive definition is simply the union of the sets of elements of the trees in the list.

An iterator is well-formed only if the trees that it contains have disjoint sets of elements. This is expressed by the inductive predicate “ok”.

```
type iterator = list (tree)

fixpoint remaining : iterator → set =
| Nil → empty
| Cons (t, ts) → elements (t) ∪ remaining (ts)

inductive ok : iterator → prop =
| ok (Nil)
| ∀ (t, ts).
   (elements (t) ∩ remaining (ts)) ≡ empty and bst (t) and ok (ts)
   ⇒ ok (Cons (t, ts))

let iterator (t) where bst (t)
returns i where (ok (i) and remaining (i) ≡ elements (t)) =
   Cons (t, Nil)

let rec next (i) where ok (i)
returns oix
where ((oix = None ⇒ remaining (i) ≡ empty)
   and (∀ (i', x). oix = Some ((i', x))
          ⇒ (remaining (i) ≡ (singleton (x) ∪ remaining (i'))
                and not (x ∈ remaining (i')) and ok (i'))))
= match i with
   | Nil → None
   | Cons (Empty, ts) → next (ts)
   | Cons (Node (h, l, x, r), ts) → Some ((Cons (l, Cons (r, ts)), x))
   end
```

Fig. 11. Iterators over binary search trees

```
let eval_cardinal (t) where bst (t)
returns n where (n = cardinal (elements (t))) =
  let rec count (i, n)
  where ok (i) and n + cardinal (remaining (i)) = cardinal (elements (t))
  returns n' where (n' = cardinal (elements (t)))
  = match next (i) with
    | None → n
    | Some ((i', x)) → count (i', n + 1)
    end
  in
    count (iterator (t), 0)
```

Fig. 12. A sample client of the iterator abstraction

```
predicate hereditary (inv, s, f) =
  ∀ (x, s', accu').
    ((s' ∪ singleton (x)) ⊆ s and not (x ∈ s') and inv (accu', s' ∪ singleton (x))) ⇒
    (pre (f) (accu', x) and (∀ accu''. (post (f) (accu', x) (accu'') ⇒ inv (accu'', s'))))

lemma hereditary_subset: ∀ (s, s', inv, f).
  (s' ⊆ s and hereditary (inv, s, f)) ⇒ hereditary (inv, s', f)

let rec fold [s, inv] (accu, t, f)
where bst (t) and elements (t) ⊆ s and inv (accu, s) and hereditary (inv, s, f)
returns accu' where inv (accu', s \ elements (t))
= match t with
  | Empty → accu
  | Node (_, l, x, r) →
    let accu_l = fold [s, inv] (accu, l, f) in
    let accu_x = f (accu_l, x) in
      fold [s \ (elements (l) ∪ singleton (x)), inv] (accu_x, r, f)
end
```

Fig. 13. Higher-order iteration over binary search trees

The function "iterator" creates an iterator "i" out of a tree "t", and satisfies the postcondition "ok(i) $\wedge$ elements(t) $\equiv$ remaining(i)", where $\equiv$ stands for extensional equality of sets (which may, or may not, coincide with definitional equality). This initial iterator is simply the singleton list [t].

The function ",", when applied to an iterator "i", returns either nothing or a pair of a new iterator "i'" and an element "x". The postcondition describes how these values are related. (The definition of the type "option", whose constructors are "None" and "Some", is omitted.)

Figure 12 shows how iterators are used. Here, the client is a function that counts the number of elements in a tree. It does not depend on the internals of the tree data structure: it only depends on the specification of iterators, which

```
let incr (x, z) returns y where (y = x + 1) = x + 1

predicate cardinal_inv (t) =
  fun (accu, s) → (accu + cardinal (s) = cardinal (elements(t)))

lemma is_hereditary_cardinal_inv:
  ∀ t. hereditary (cardinal_inv (t), elements (t), incr)

let eval_cardinal (t) where bst (t)
returns x where (x = cardinal (elements (t)))
= fold [elements (t), cardinal_inv (t)] (0, t, incr)
```

Fig. 14. A sample client of the fold operator

is expressed in terms of abstract (logical-level) sets. So, this client code could be placed in another module, without access to the definition of trees.

The "eval_cardinal" function performs a loop, expressed as an internal recursive function, with an integer accumulator n. It corresponds directly to a **foreach** construct in Java or C#. The precondition of this internal function represents the loop invariant: the number of elements counted so far, plus the number of elements remaining to be seen, equals the total number of elements of the set. The postcondition is simply the precondition, specialized to the case where no elements remain.

The precondition of "count" must also state that "i" is an "ok" iterator, even though it does not have to know about the definition of "ok". This is somewhat undesirable. In the future, we will want to allow defining a dependent sum type of the form "i : iterator **where** ok(i)", and exporting it as an abstract type.

The definition of "eval_cardinal" is syntactically somewhat heavy, as it is expressed in our core language. In a full-scale programming language, a more palatable syntax for loops could be introduced, and desugared into recursive functions and iterators. A single formula, the loop invariant, would have to be written down, instead of two formulae in this low-level version of the code.

7.5 Higher-Order Iteration

We now present a specification of the classic "fold" higher-order function over sets implemented as binary search trees. The specification is rather more complex than that of first-order iterators, for at least two reasons. First, the specification must mention the client's state (the accumulator) and invariant. Second, because the code is not tail-recursive, some information is implicitly encoded within the stack, and a ghost parameter is used to make it explicit in the specification.

The function "fold" is parameterized over two ghost variables, namely the client invariant "inv" and a set "s" of remaining elements. In the case of first-order iterators, the former was unnecessary because the client retains control over the desired invariant, and the latter was unnecessary because the set of remaining elements was directly expressed as "remaining(i)". Here, the set of remaining elements is implicit in the stack, so a ghost variable must be used in order to refer to it.

The precondition of “fold” expresses the following requirements. First, “t” must a binary search tree. Second, the elements of “t” must form a subset of “s”. This reflects that, in general, “t” is a sub-tree of a larger tree over which iteration is taking place. Third, the invariant must initially hold. Last, the invariant must be hereditary: that is, at any time, if an element “x” is picked among the remaining elements, the invariant guarantees that it is legal to apply “f” to the current accumulator and to “x”, and guarantees that the new accumulator thus obtained will still satisfy the invariant.

This definition is certainly somewhat overwhelming. It shows, at the same time, that it is possible to specify and exploit higher-order functions in our framework, and that there is a cost in complexity to be paid for doing so. More experience is needed before we can tell how easily higher-order functions can be defined and used in practice.

7.6 Quantitative Results

The binary search tree library contains 22 functions. The development is composed of 108 lemmas, 603 lines of specification and 247 lines of code. The factor of 3 in size between specification and code does not necessarily mean that specifications must be heavy: in a realistic system, a large part of the specification would be imported from a standard library. 749 proof obligations are generated and are proven automatically by Alt-Ergo [23]. Only one lemma, stating that the height of a tree is nonnegative, requires an induction in Coq [22]; the other lemmas are proven automatically by Alt-Ergo. About 80% of the proofs require less than 5 seconds to be proven by Alt-Ergo. Yet, about 10% of the proofs require from 10 to 30 minutes. A forthcoming extension of Alt-Ergo with support for reasoning modulo associativity and commutativity of some set operations (such as set union) would perhaps improve these results.

8 Related Work

The roots of our work lie in Hoare logic [1, 2]. Extensions of Hoare logic with support for recursive, higher-order procedures were heavily studied in the late 1970’s and early 1980’s [4, 5, 6, 7, 8]. In particular, the issue of completeness received a lot of attention after Clarke [4] proved that there can be no sound and complete Hoare logic for a programming language equipped with recursive, higher-order procedures and global variables. Clarke’s result, however, is based upon the assumption that formulae and proof obligations are expressed in a first-order logic. Damm and Josko [6] point out that, by moving to higher-order logic, it is possible to work around Clarke’s negative result. In this paper, we follow Damm and Josko and allow specifications to be expressed in higher-order logic. The intuitive justification for this approach is that, if functions can abstract over functions, then specifications must abstract over specifications.

Our work has been strongly inspired by several existing, practical tools for checking imperative programs [10, 11, 12, 13, 41, 42]. This paper is an attempt to exploit the strengths of these works while steering away from imperative programming and placing renewed emphasis on polymorphism and modularity.

Our method for generating proof obligations is particularly straightforward: it appears in its entirety in Figure 8. In comparison with the method used in ESC/Java [43], we avoid a translation to "passive form" because we have no assignments to begin with. We avoid the exponential explosion that could follow from the interplay between sequences and alternatives by requiring sequences (that is, **let** constructs) to carry user-provided postconditions (§3.3).

Our system is not sound with respect to a call-by-name dynamic semantics. There are at least two reasons for this fact. First, some divergent expressions admit **false** as a valid postcondition. If such an expression e_1 is made the first component of a sequence, as in "$\mathbf{let}\, x/\mathbf{false} = e_1 \;\mathbf{in}\; e_2$", then second component e_2 is checked under the assumption **false**. As a result, all of the the proof obligations found within e_2 are vacuously satisfied. This is sound under call-by-value evaluation, because e_2 is never executed. It is unsound under call-by-name evaluation, because e_2 is executed immediately (after binding x to a suspension). The second reason is that, in a call-by-name semantics, every type is inhabited by a bottom value, and some types are inhabited by infinite values. This is not reflected in the way we lift computational values and types up to the logical level.

Scott's logic of computable functions [44] interprets λ-terms in a denotational model, where equality implies, or coincides with, observational equivalence. It comes with a set of sound deduction rules, and allows explicit reasoning about divergence and equality of computations. It admits call-by-value and call-by-name variants. It was implemented as early as 1972 by Milner [45]. More recent implementations [46, 47, 48] embed Scott's LCF within some form of higher-order logic. In a somewhat similar vein, Longley and Pollack [49] embed the functional core of Standard ML, via a fully abstract denotational semantics, into higher-order logic.

Our approach is less elaborate: by focusing on partial correctness, by adopting a call-by-value semantics, and by lifting only values, as opposed to expressions, up to the logical level, we are able to ignore non-termination issues entirely, and to work with value spaces that do not have bottom elements or definedness orderings. By contrast, tools or approaches that focus on lazy functional programs, such as Programatica [50, 51] or the Cover translator [52], require reasoning about non-termination, resulting in proof obligations that can become cluttered with definedness side conditions. The simplicity of our approach comes at a cost: our system can neither establish termination of an expression nor reason about observational equality of expressions.

Honda and Yoshida [53] define a Hoare logic for call-by-value higher-order functions, to which our system seems rather analogous. A technical difference is that Honda and Yoshida allow expressions (including, in particular, function applications) to appear within formulae, and interpret equality as observational equality; whereas we only lift values to the logical level, and interpret equality as equality of values. Honda and Yoshida's system does not seem to have been implemented.

Smith [54, §4.4.1] defines a type theory with partial objects, where the type $\bar{A}$ contains the possibly non-terminating computations that yield a result of type A. Smith notes that the fixed point axiom, which has type $(A \to A) \to A$, is sound only at *admissible* types. As an example of a non-admissible type, he offers a type D whose definition can be read: "D is the type of the partial functions g of naturals to naturals such that g diverges for at least one input". It is easy to construct a function of type $D \to D$ whose least fixed point is in fact a total function: this shows that D is not admissible. A reviewer of an earlier version of the present paper noted that "g diverges for at least one input" seems expressible, in our system, as $\exists x.\forall y.\neg \mathbf{post}(g)(x)(y)$, and wondered if Smith's example could be adapted to show that our system is unsound. One should note, first, that although this formula indeed represents a *sufficient* condition for g to diverge for at least one input, it is not a *necessary* condition. Indeed, the predicate $\mathbf{post}(g)$ denotes the programmer-provided postcondition of g; it does not denote the actual semantics of g. Second, when the programmer supplies an explicit definition of the predicate $\mathbf{post}(g)$ (which he must do), this definition cannot refer to g itself. As a result, there is no way that the postcondition associated with g can be the self-referent "g diverges for at least one input".

ESC/Haskell [55] allows annotating Haskell programs with preconditions and postconditions that are also expressed in Haskell. A special-purpose theorem prover, based on symbolic evaluation of Haskell terms, is developed.

The theorem prover Coq [22] can be used as a programming language, in which programs are both developed and proved correct. The Compcert certified compiler [56] offers an example of a large program developed in this style. However, there is some agreement that Coq is not (yet) a convenient programming language: for instance, it only allows writing pure, terminating functions.

The programming language Russell [38] extends Coq with facilities for defining programs annotated with assertions, in the style of Hoare logic. There are many similarities between Russell and our work. One important technical difference is that we separate the typechecking process, which is performed first and remains traditional, and the process of extracting proof obligations, which runs as a second phase, whereas, in Russell, as in Coq, typechecking and proving are one and the same activity. In particular, Russell encourages the use of indexed types, like *list* n, so that typechecking can give rise to proof obligations: for instance, supplying an actual argument of type *list* m to a function that expects a formal parameter of type *list* n generates the proof obligation $m = n$. Another difference is that Russell terms are elaborated into Coq terms, whereas we adopt a less foundational approach and are happy to trust an external theorem prover.

Hoare Type Theory [33, 57] is somewhat similar to our system, insofar as it offers decidable basic typechecking and decidable generation of proof obligations. It also shares our use of higher-order logic and our emphasis on polymorphism and abstraction. It is much more ambitious than our proposal, in that it attempts to deal not only with algebraic data types and higher-order functions, but also with heap-allocated, mutable state. As a result, its design and metatheory are considerably more involved.

Some authors [33, 55, 58, 59] allow code to appear in specifications. This is motivated partly by a desire to make formulae executable, so as to allow assertions to be checked at runtime, and partly by fear that, otherwise, a single functionality might have to be implemented twice: once at the computational level, once at the logical level. Our technical and philosophical choice is different: we consider all code as potentially impure, and do not allow code to appear within specifications. We do not check assertions at runtime: if the programmer wishes to insert a runtime check, she must do so explicitly. Furthermore, we believe that, in practice, opportunities for code sharing between computational and logical levels are rare: the oft-cited case of lists is one of only a few situations where implementation and specification coincide.

Indexed types [26, 60] and refinement types [61] rely on so-called indices. Indices are elements of some mathematical domain, such as an arbitrary finite set, or the set of all natural numbers. Types are enriched with constraints over indices, allowing invariants, preconditions, and postconditions to be expressed. The syntax of constraints is carefully restricted so as to ensure that constraint entailment is decidable. This allows proof obligations to be automatically checked. Generalized algebraic data types [39] are also an instance of this idea, where indices are types, that is, first-order terms. The appeal of this approach resides in the high degree of automation that it allows. On the other hand, this comes at the price of a restriction to a decidable logic. In fact, our decision of using a highly expressive, hence undecidable, logic was motivated by our earlier study of generalized algebraic data types [62, 63].

Going beyond indexed types, several programming languages offer full dependent types [37, 64, 65, 66]. By exploiting the Curry-Howard isomorphism, they allow code and proofs to be expressed and combined within a single language. This allows programs to appear more self-contained, but means that a fragment of the programming language must be a consistent logic, and requires mechanisms to assist the user in building proofs. Our design, which relies on an off-the-shelf theorem prover, is more modular.

9 Conclusion

We have presented a simple methodology for extracting proof obligations out of call-by-value functional programs. Our proposed future work includes:

- extending our prototype implementation [21] and equipping it with a compilation path down to Objective Caml;
- relaxing our positivity condition (§3.2), which restricts the use of functions within data structures, preventing, for instance, the standard definition of infinite streams;
- internalizing type equality, that is, introducing equations between types into the syntax of formulae, together with suitable conversion rules for exploiting

such equations; indeed, we, and other authors [33], have noticed that such an extension would subsume generalized algebraic data types [39];
– studying the issues raised by modularity and mutable state.

Acknowledgement. The authors wish to thank the anonymous reviewers of a previous version of this paper for contradicting a false claim and offering useful comments and suggestions. Thanks are also due to Sylvain Conchon and Evelyne Contejean for their great work on Alt-Ergo [23] that helped us demonstrate our approach practically.

References

1. Floyd, R.W.: Assigning meanings to programs. In: Mathematical Aspects of Computer Science. Proceedings of Symposia in Applied Mathematics, vol. 19, pp. 19–32. American Mathematical Society (1967)
2. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12(10), 576–580 (1969)
3. Cousot, P.: Methods and logics for proving programs. In: Formal Models and Semantics. Handbook of Theoretical Computer Science, vol. B, pp. 841–993. Elsevier Science, Amsterdam (1990)
4. Clarke, E.: Programming language constructs for which it is impossible to obtain good Hoare axiom systems. Journal of the ACM 26(1), 129–147 (1979)
5. Apt, K.R.: Ten years of Hoare's logic: A survey—part I. ACM Transactions on Programming Languages and Systems 3(4), 431–483 (1981)
6. Damm, W., Josko, B.: A sound and relatively* complete axiomatization of Clarke's language L4. In: Clarke, E., Kozen, D. (eds.) Logic of Programs 1983. LNCS, vol. 164, pp. 161–175. Springer, Heidelberg (1984)
7. German, S., Clarke, E., Halpern, J.: Reasoning about procedures as parameters. In: Clarke, E., Kozen, D. (eds.) Logic of Programs 1983. LNCS, vol. 164, pp. 206–220. Springer, Heidelberg (1984)
8. Goerdt, A.: A Hoare calculus for functions defined by recursion on higher types. In: Parikh, R. (ed.) Logic of Programs 1985. LNCS, vol. 193, pp. 106–117. Springer, Heidelberg (1985)
9. Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer 7(3), 212–232 (2005)
10. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: ACM Conference on Programming Language Design and Implementation (PLDI), pp. 234–245 (2002)
11. Marché, C., Paulin-Mohring, C., Urbain, X.: The Krakatoa tool for certification of Java/JavaCard programs annotated in JML. Journal of Logic and Algebraic Programming 58(1–2), 89–106 (2004)
12. Filliâtre, J.C., Marché, C.: Multi-prover Verification of C Programs. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 15–29. Springer, Heidelberg (2004)
13. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# Programming System: An Overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)

14. Hughes, J.: Why functional programming matters. Computer Journal 32(2), 98–107 (1989)
15. Mehta, F., Nipkow, T.: Proving pointer programs in higher-order logic. Information and Computation 199(1–2), 200–227 (2005)
16. Detlefs, D.L., Leino, K.R.M., Nelson, G.: Wrestling with rep exposure. Research Report 156, SRC (July 1998)
17. Leino, K.R.M., Nelson, G.: Data abstraction and information hiding. ACM Transactions on Programming Languages and Systems 24(5), 491–553 (2002)
18. Fähndrich, M., DeLine, R.: Adoption and focus: practical linear types for imperative programming. In: ACM Conference on Programming Language Design and Implementation (PLDI), pp. 13–24 (June 2002)
19. Clarke, D.G., Potter, J.M., Noble, J.: Ownership types for flexible alias protection. In: ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 48–64 (October 1998)
20. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: IEEE Symposium on Logic in Computer Science (LICS), pp. 55–74 (2002)
21. Régis-Gianas, Y.: A Hoare logic for call-by-value functional programs: Prototype tool (January 2008), `http://pangolin-programming-language.googlecode.com`
22. The Coq development team: The Coq Proof Assistant (2006)
23. Conchon, S., Contejean, E.: The Alt-Ergo automatic theorem prover (2006), `http://alt-ergo.lri.fr/`
24. Kaplan, H., Tarjan, R.E.: Purely functional, real-time deques with catenation. Journal of the ACM 46(5), 577–603 (1999)
25. Milner, R.: A theory of type polymorphism in programming. Journal of Computer and System Sciences 17(3), 348–375 (1978)
26. Xi, H.: Dependent Types in Practical Programming. PhD thesis, Carnegie Mellon University (December 1998)
27. Leroy, X., Doligez, D., Garrigue, J., Rémy, D., Vouillon, J.: The Objective Caml system (October 2005)
28. Andrews, P.B.: An introduction to mathematical logic and type theory: to truth through proof. Academic Press, London (1986)
29. Paulin-Mohring, C.: Inductive definitions in the system Coq: rules and properties. Research Report RR1992-49, ENS Lyon (1992)
30. Werner, B.: Une Théorie des Constructions Inductives. PhD thesis, Université Paris 7 (1994)
31. Flanagan, C., Sabry, A., Duba, B.F., Felleisen, M.: The essence of compiling with continuations. In: ACM Conference on Programming Language Design and Implementation (PLDI), pp. 237–247 (1993)
32. Régis-Gianas, Y.: Des types aux assertions logiques: preuve automatique ou assistée de propriétés sur les programmes fonctionnels. PhD thesis, Université Paris 7 (November 2007)
33. Nanevski, A., Ahmed, A., Morrisett, G., Birkedal, L.: Abstract Predicates and Mutable ADTs in Hoare Type Theory. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 189–204. Springer, Heidelberg (2007)
34. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. Journal of the ACM 52(3), 365–473 (2005)
35. Lescuyer, S.: Codage de la logique du premier ordre polymorphe multi-sortée dans la logique sans sortes. Master's thesis, Master Parisien de Recherche en Informatique (2006)
36. Kerber, M.: How to prove higher order theorems in first order logic. In: International Joint Conferences on Artificial Intelligence, pp. 137–142 (1991)

37. Altenkirch, T., McBride, C., McKinna, J.: Why dependent types matter (unpublished) (April 2005)
38. Sozeau, M.: Subset coercions in Coq. In: TYPES (2006)
39. Xi, H., Chen, C., Chen, G.: Guarded recursive datatype constructors. In: ACM Symposium on Principles of Programming Languages (POPL), pp. 224–235 (January 2003)
40. Filliâtre, J.C.: Backtracking iterators. In: ACM Workshop on ML (September 2006)
41. Detlefs, D.L., Leino, K.R.M., Nelson, G., Saxe, J.B.: Extended static checking. Research Report 159, Compaq SRC (December 1998)
42. Filliâtre, J.C.: Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud (March 2003)
43. Flanagan, C., Saxe, J.B.: Avoiding exponential explosion: generating compact verification conditions. In: ACM Symposium on Principles of Programming Languages (POPL), pp. 193–205 (2001)
44. Scott, D.S.: A type-theoretical alternative to ISWIM, CUCH, OWHY. Theoretical Computer Science 121(1–2), 411–440 (1993)
45. Milner, R.: Implementation and applications of Scott's logic for computable functions. In: Proceedings of the ACM conference on proving assertions about programs, pp. 1–6 (January 1972)
46. Agerholm, S.: A HOL basis for reasoning about functional programs. Technical Report RS-94-44, BRICS (December 1994)
47. Bartels, F., von Henke, F., Pfeifer, H., Rueß, H.: Mechanizing domain theory. Ulmer Informatik-Berichte 96-10, Universität Ulm, Fakultät für Informatik (1996)
48. Müller, O., Nipkow, T., von Oheimb, D., Slotosch, O.: HOLCF = HOL + LCF. Journal of Functional Programming 9, 191–223 (1999)
49. Longley, J., Pollack, R.: Reasoning About CBV Functional Programs in Isabelle/HOL. In: Slind, K., Bunker, A., Gopalakrishnan, G.C. (eds.) TPHOLs 2004. LNCS, vol. 3223, pp. 201–216. Springer, Heidelberg (2004)
50. Kieburtz, R.B.: P -logic: Property verification for Haskell programs. Draft (August 2002)
51. Hallgren, T., Hook, J., Jones, M.P., Kieburtz, R.: An overview of the Programatica toolset. In: High Confidence Software and Systems Conference (HCSS) (2004)
52. Abel, A., Benke, M., Bove, A., Hughes, J., Norell, U.: Verifying Haskell programs using constructive type theory. In: Haskell workshop, pp. 62–73 (September 2005)
53. Honda, K., Yoshida, N.: A compositional logic for polymorphic higher-order functions. In: International ACM Conference on Principles and Practice of Declarative Programming (PPDP), pp. 191–202 (August 2004)
54. Smith, S.F.: Partial Objects in Type Theory. PhD thesis, Cornell University (January 1989)
55. Xu, D.N.: Extended static checking for Haskell. In: Haskell workshop, pp. 48–59. ACM Press, New York (2006)
56. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: ACM Symposium on Principles of Programming Languages (POPL), pp. 42–54 (January 2006)
57. Nanevski, A., Morrisett, G., Birkedal, L.: Polymorphism and separation in Hoare type theory. In: ACM International Conference on Functional Programming (ICFP), pp. 62–73 (September 2006)
58. Barnett, M., Naumann, D.A., Schulte, W., Sun, Q.: 99.44% pure: Useful abstractions in specifications. In: Formal Techniques for Java-like Programs (2004)

59. Gronski, J., Knowles, K., Tomb, A., Freund, S.N., Flanagan, C.: Sage: Hybrid checking for flexible specifications. In: Scheme and Functional Programming (September 2006)
60. Zenger, C.: Indexed types. Theoretical Computer Science 187(1–2), 147–165 (1997)
61. Davies, R.: Practical refinement-type checking. Technical Report CMU-CS-05-110, School of Computer Science, Carnegie Mellon University (May 2005)
62. Pottier, F., Régis-Gianas, Y.: Towards efficient, typed LR parsers. In: ACM Workshop on ML. Electronic Notes in Theoretical Computer Science, vol. 148(2), pp. 155–180 (March 2006)
63. Pottier, F., Régis-Gianas, Y.: Stratified type inference for generalized algebraic data types. In: ACM Symposium on Principles of Programming Languages (POPL) (January 2006)
64. Chen, C., Xi, H.: Combining programming with theorem proving. In: ACM International Conference on Functional Programming (ICFP) (September 2005)
65. Sheard, T.: Putting Curry-Howard to work. In: Haskell workshop (2005)
66. Westbrook, E., Stump, A., Wehrman, I.: A language-based approach to functionally correct imperative programming. In: ACM International Conference on Functional Programming (ICFP), pp. 268–279 (2005)