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.

Categories and Subject Descriptors D.3.3 [Programming Languages]: Language Constructs and Features—Data types and structures; Polymorphism; Procedures, functions, and subroutines; Recursion; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs—Assertions; Mechanical verification

General Terms Theory

Keywords Hoare logic, extended static checking

1. Introduction
Hoare logic [24, 30, 14] 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. The process of constructing and checking proof obligations is sometimes known as “extended static checking” [19].

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 [12, 5, 15, 26, 27]. 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, C [21], Java [9, 22, 41], and C# [6].

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 [32]. 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 [42, 21].

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 [18, 36]. In order to address this issue, researchers have developed linear types and regions [25], ownership types [11], and separation logic [51], 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.
Today, to the best of our knowledge, no tool exists that can extract proof obligations out of a call-by-value functional program and pass them on to an off-the-shelf, interactive or automated, theorem prover. (See §8 for a more precise discussion of related work.) We would like to fill this gap.

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 prototype tool [52] has been developed, which works in conjunction with the interactive theorem prover Coq [56], with the automated first-order theorem prover Ergo [13], 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 persistent double-ended queues [33]. 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 $\text{true}$ 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 [44] is possible, and implemented in our tool [52]. Type inference does not generate any proof obligations. We do not have dependent types, such as lists indexed with an integer length [58], but simulate them as follows. Instead of declaring that $x$ has type $\text{list}\ n$, we declare that $x$ has type $\text{list}$, and assert the logical formula $\text{length}(x) = n$, where the function $\text{length}$ is inductively defined at the logical level.

Formulæ 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 formulæ. But what does it mean for a formula to refer to a computational function $f$ of type, say, $\tau_1 \rightarrow \tau_2$? 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) $(\tau_1 \rightarrow \text{prop}) \times (\tau_1 \rightarrow \tau_2 \rightarrow \text{prop})$. The two pair projections, written $\text{pre}$ and $\text{post}$, can be used to refer to the pair components. That is, $\text{pre}(f)$ and $\text{post}(f)$ 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 formulæ. However, when $f$ is unknown ($\lambda$- or V-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 [38], 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 [4] whose types and terms appear in Figure 1. Types $\theta$ include type variables, pa-
Parameterized inductive types, function types, product types, and the type prop of logical propositions. In the following, the syntax of types is extended with standard syntactic sugar for falsity, disjunction, implication, equivalence, existential quantification, etc. 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 \( \alpha \) and every use of a variable \( x \) is explicitly annotated with a type variable \( \theta \), which states how the type scheme associated with \( x \) is instantiated. For this reason also, the syntax of formulae offers universal quantification over type variables, so a fact can be asserted to hold at all types.

The logic offers parameterized inductive types. We assume that each inductive type constructor \( d \) carries a fixed integer arity, and that every application \( d \theta \) 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:

\[
\forall \alpha, \theta_1 \times \ldots \times \theta_n \rightarrow d \theta
\]

We impose a positivity condition [48], 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.

### Computational Types

\[
\begin{align*}
\tau & ::= \alpha \\
& | \bar{\tau} \\
& | \tau \times \tau \\
\sigma & ::= \forall \alpha, \tau
\end{align*}
\]

### Computational Type Environments

\[
\begin{align*}
\Gamma & ::= \emptyset \\
& | \bar{\alpha} \sigma \\
& | \Gamma, (x : \sigma) \\
& | \Gamma, \bar{\alpha}
\end{align*}
\]

### Values

\[
\begin{align*}
v & ::= x \bar{\tau} \\
& | D \bar{\tau}(v, \ldots, v) \\
& | \text{fun } f(x : \tau/F) : (x : \tau/F) = e
\end{align*}
\]

### Patterns

\[
\begin{align*}
p & ::= x \bar{\tau} \\
& | D \bar{\tau}(p, \ldots, p)
\end{align*}
\]

### Expressions

\[
\begin{align*}
e & ::= v \\
& | v(v) \\
& | \text{let } (x \bar{\alpha} : \tau/F) = e \text{ in } e \\
& | \text{case } v \text{ of } c
\end{align*}
\]

### Cases

\[
\begin{align*}
c & ::= \emptyset \\
& | (p \leftrightarrow e) \bar{c}
\end{align*}
\]
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 $\vec{\alpha}$ can be
explicitly bound.

A function definition takes the general form:

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

The keyword `/` 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, $\text{true}$ is assumed).

A local variable definition takes the general form:

$$\text{let} \ (x \ : \ \tau / F) = e_1 \ \text{in} \ e_2$$

The local variable $x$ is bound within $F$ and within $e_2$. The type vari-
ables $\vec{\alpha}$ are bound within $\tau$, $F$, and $e_1$. The proposition $F$ serves
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:

$$\text{case} \ v \ of \ c$$

Here, $c$ is a possibly empty sequence of cases (i.e., branches). Each branch
is of the form $(p \rightarrow e)$, where the variables that appear in
the pattern $p$ are bound within $e$. Patterns must be linear.

### 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 $\tau$, 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 $F$?

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 $\tau$, we associate a logical type $\pi \ \tau$, and to each computational value $v$, we associate a logical term $\pi \ v$, with the intended property that if $v$ has computational type $\tau$, then $\pi \ v$ has logical type $\pi \ \tau$.

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:

$$\pi \tau_1 \rightarrow \pi \tau_2 = (\pi \tau_1 \rightarrow \text{prop}) \times (\pi \tau_1 \rightarrow \pi \tau_2 \rightarrow \text{prop})$$

The first component of the pair, which represents the function’s pre-
condition, is abstracted over the function’s argument, while the sec-
ond 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 $\tau_1 \rightarrow \tau_2$, then a formula embedded
within the code, in the scope of $f$, views $f$ as a pair of predicates,
and can refer to $\text{pre} \ f$ and $\text{post} \ f$. (Recall that, as per Figure 1,
$\text{pre}$ and $\text{post}$ are sugar for the projections $\pi_1$ and $\pi_2$.)

Values of computational function type (that is, $\lambda$-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 7.) This
reflects a philosophy in which the only way of reasoning about the
behavior of a function is via its specification: it is not possible, in
our approach, to reason directly about the code of a function.

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

$$D : \forall \vec{\alpha}, \tau_1 \times \ldots \times \tau_n \rightarrow d \vec{\alpha},$$

there must be a logical-level data constructor

$$D : \forall \vec{\alpha}, \tau_1 \ldots \tau_n \rightarrow d \vec{\alpha}.$$  

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 post-
conditions for every construct in our programming language.
For instance, the strongest postcondition for a value $v$ is $\lambda x. (x = \pi \ v)$.
The strongest postcondition for a function application $v_1(v_2)$ is $\pi \ \text{post} \ (v_1)(v_2)$. 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 exist-
tential 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 $\text{true}$,
otherwise.

It is possible, via a conversion to $\lambda$-normal form, to ensure that the
left-hand sub-expression of a `let` construct is never another `let`
construct. So, 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 they are kept
around in order to make a subject reduction statement and prove
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 $\alpha$ 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 $\alpha \rightarrow \tau$ for the substitution that replaces
every free occurrence of $\alpha$ at the computational level with $\tau$ and
every free occurrence of $\alpha$ at the logical level with $\pi \ \tau$. Similarly,
we write $x \rightarrow v$ 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 $\pi \ v$. 
We have annotated \texttt{let} constructs with explicit type abstractions and occurrences of variables with explicit type applications. As a result, contracting a \( [d \bar{\tau}] \) allows. The formal definition is:

\[
\text{let } (x \to \alpha : \tau) / F = v \text{ in } e \rightarrow [x \mapsto v][f \mapsto v_1]e \quad \text{if } v_1 \text{ is } \text{fun } f (x : \tau / F) : (\ldots) = e
\]

\text{case } v \text{ of } (p \mapsto e) \mid c \rightarrow \text{case } v \text{ of } e \quad \text{if } [p \mapsto v] \text{ is defined}

\text{let } (x \to \alpha : \tau / F) = e_1 \text{ in } e_2 \rightarrow \text{let } (x \to \alpha : \tau / F) = e'_1 \text{ in } e_2 \quad \text{if } e_1 \to e'_1

\[
\begin{array}{ll}
F \text{ valid} & \Gamma \models \forall (x : [\sigma], F) \models F \\
\emptyset \models F & \Gamma, (x : [\sigma]) \models F \\
\Gamma, \alpha \models F & \Gamma, F_1 \models F_2
\end{array}
\]

\[
\begin{array}{ll}
\emptyset \models F & \Gamma \models \forall (x : [\sigma], F) \models F \\
\Gamma, (x : [\sigma]) \models F & \Gamma, \alpha \models F \\
\Gamma, F_1 \models F_2
\end{array}
\]

In practice, our tool [52] 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 \( \Gamma \) 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 \( \Gamma, F \) is well-formed when \( F \) has type \text{prop} under \( [\Gamma] \).

4.2 Proof obligations

A proof obligation is a judgement of the form \( \Gamma \models F \), where \( F \) has type \text{prop} under \( [\Gamma] \). This judgement is defined by the rules of Figure 6. When read from bottom to top, the rules can be understood as translating a proof obligation to a closed formula of type \text{prop}, which must be valid for the judgement to hold. The validity of a formula 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:
Figure 9. The computation language (proof system: patterns)

\[
\begin{align*}
(\Gamma, (x : \sigma))(x) &= \sigma \\
(\Gamma, (x_1 : \sigma_1)) (x_2) &= \Gamma(x_2) \text{ if } x_1 \neq x_2 \\
(\Gamma, \alpha)(x) &= \Gamma(x) \text{ if } \alpha \neq \Gamma(x) \\
(\Gamma, F)(x) &= \Gamma(x)
\end{align*}
\]

**VAR**
- \(\Gamma(x) = \forall \alpha. \tau\)
- \(\Gamma \vdash x : [\alpha \mapsto \tau]\)

**DATA**
- \(\Gamma \vdash \forall \alpha. \tau_1 \times \ldots \times \tau_n \rightarrow d \alpha\)
- \(\forall \alpha \vdash \forall \alpha. \tau_1 \rightarrow \tau_2\)
- \(\Gamma, \forall \alpha \vdash v_1 : [\alpha \mapsto \tau] \vdash v_2 : [\alpha \mapsto \tau] \vdash \forall \alpha. \tau_1 \rightarrow \tau_2\)
- \(\Gamma \vdash D \tau (v_1, \ldots, v_n) : d \tau\)

**FUN**
- \(\Gamma \vdash f \neq F_1, F_2\)
- \(\Gamma \vdash F_1 \vdash \text{prop}\)
- \(\Gamma, (f \vdash \tau_1 \rightarrow \tau_2), F \vdash \text{fun} f \ldots, (x_1 : \tau_1), F_1 \vdash e : \tau_2 \vdash \lambda (x_2 : [\tau_2]), F_2\)
- \(\Gamma \vdash \text{fun} f(x_1 / F_1) : (x_2 : \tau_2) F_2 = e : \tau_1 \rightarrow \tau_2\)

**PAT-VAR**
- \((x : \tau) \vdash x : \tau\)

**PAT-DATA**
- \(\Gamma \vdash \forall \alpha. \tau_1 \times \ldots \times \tau_n \rightarrow d \alpha\)
- \(\forall \alpha \vdash \forall \alpha. \tau_1 \rightarrow \tau_2\)
- \(\Gamma, \forall \alpha \vdash v_1 : [\alpha \mapsto \tau] \vdash v_2 : [\alpha \mapsto \tau] \vdash \forall \alpha. \tau_1 \rightarrow \tau_2\)
- \(\Gamma \vdash D \tau (v_1, \ldots, v_n) : d \tau\)

**4.4 Values**
The judgement \(\Gamma \vdash v : \tau\) (Figure 7) states that, under the type environment \(\Gamma\), the value \(v\) has type \(\tau\). 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 \([v]\).

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, which spans two lines, 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

\[ f = [\text{fun} f \ldots], \]

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

\[ f = (\lambda (x_1 : [\tau_1]). F_1, F_1, (x_2 : [\tau_2]), F_2). \]

This hypothesis gives meaning to occurrences of \(\text{prefix}(f)\) and \(\text{postfix}(f)\) 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 \(\lambda (x_2 : [\tau_2]), F_2\).

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

**4.5 Patterns**
The judgement \(\Gamma \vdash p : \tau\) (Figure 8) states that a value of type \(\tau\) can safely be matched against the pattern \(p\), giving rise to (exactly) the bindings described by \(\Gamma\). 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.

**4.6 Expressions**
The judgement \(\Gamma \vdash e : \tau\{P\}\) (Figure 9) states that, under the type environment \(\Gamma\), the expression \(e\) has type \(\tau\) and (if it terminates) produces a value whose logical reflection satisfies the predicate \(P\). In such a judgement, \(P\) has type \(\tau \rightarrow \text{prop}\) under \(\Gamma\).

Rule **Value** directly reflects this intended meaning: the judgement \(\Gamma \vdash v : \tau\{P\}\) holds if and only if \(v\) has type \(\tau\) under \(\Gamma\) and its logical reflection \([v]\) provably satisfies \(P\) under the hypotheses found in \(\Gamma\). The premise \(\Gamma \vdash P\{[v]\}\) 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 \(\tau_2 \rightarrow \text{prop}\), for \(\forall (x : [\tau_2]) . (P'(x) \Rightarrow P(x))\), where \(x\) is fresh for \(P'\) and \(P\).

Rule **Let** checks that \(e_1\) has type \(\tau_1\) and that \(e_2\) complies with the postcondition \(F\). Then, the rule performs type generalization, in the style of Milner [44], so that \(e_2\) is checked under the assignment \((x : [\forall \alpha. \tau_1])\). The hypothesis \(F\) is changed into \(\forall \alpha. [x \mapsto x:F]\), so as to reflect the fact that \(x\) now has polymorphic type.

Rule **Case-Nil** emits the proof obligation \(\Gamma \vdash \text{false}\), which requires the conjunction of hypotheses found within \(\Gamma\) 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 \(\tau\). The environment \(\Gamma'\) 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 \( \Gamma \) with \( \Gamma' \) and with the hypothesis \( [v] = [p] \), the branch \( e \) must have the desired type \( \tau \) and meet the desired postcondition \( P \). Furthermore, under the hypothesis that no instance of \( p \) matches \( v \), which is written \( \forall \Gamma', [v] \neq [p] \), the remaining branches must have type \( \tau \) and meet the postcondition \( P \). (Our use of \( [p] \) plays on 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 \( \forall \Gamma', [v] \neq [p_i] \) 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 and 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 assert, which requires false to hold, marks a piece of code as inaccessible. Here, it can be viewed as 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

\[
\text{let logic } x : \theta/F \text{ in } e
\]

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, computational occurrences of \( x \) 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:

\[
\text{fun } f(\text{logic } x : \theta(x_1 : \tau_1) : (x_2 : \tau_2) = e
\]

This construct binds \( x \) within \( F_1, F_2 \), and \( e \). Note that \( \theta \) 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 [46]. In a realistic implementation, however, they should be primitive notions.

5. A few extensions

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

\[
\text{assert } F \text{ in } e
\]

This construct requires \( F \) to hold: a proof obligation is emitted. It has no computational content: dynamically, it behaves like \( e \). Here, it can be viewed as syntactic sugar for a let construct. 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 requires false to hold, marks a piece of code as inaccessible. Here, it can be viewed as syntactic sugar for a case construct with zero branches.

6. Interfacing with external theorem provers

6.1 Coq

Our typed, higher-order logic is easily embedded within the Calculus of Inductive Constructions, which underlies Coq [56]. As a result, exporting proof obligations towards Coq is a simple matter of pretty-printing.

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 Ergo

Ergo [13] is a full-fledged automated theorem prover for a typed, polymorphic, first-order logic. Its design is partly inspired by Simplify [17]. However, Ergo’s logic is typed and polymorphic, whereas Simplify’s is untyped. This makes 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 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 [39], but are complex and costly. Of course, the trivial encoding that erases all types is unsound.

In addition to first-order logic, 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 $\beta$-conversion [34].

Perhaps surprisingly, in our case, this encoding can be made to look fairly natural. The symbols $\text{pre}$ and $\text{post}$, 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 $\text{pre}$ a binary predicate and $\text{post}$ a ternary predicate, avoiding curried function applications. That is, instead of the higher-order formula:

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

we can write:

$$\forall(x_1 : [\tau_1]).(\text{pre}(f, x_1) \iff F_1) \land \forall(x_1 : [\tau_1]).\forall(x_2 : [\tau_2]).(\text{post}(f, x_1, x_2) \iff F_2)$$

The pair and the three $\lambda$-abstractions have been $\eta$-expanded, and the projection and application symbols have been fused into applications of $\text{pre}$ and $\text{post}$. 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 computable types is modified so that the computable function type constructor is no longer interpreted:

$$\rightarrow \tau_1 \rightarrow \tau_2 = [\tau_1] \rightarrow [\tau_2]$$

That is, we make $\rightarrow$ 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 $\tau_1 \rightarrow \tau_2$. The type schemes assigned to $\text{pre}$ and $\text{post}$ are as follows:

$$\text{pre} : \forall\alpha_1\alpha_2.(\alpha_1 \rightarrow \alpha_2) \times \alpha_1 \rightarrow \text{prop}$$

$$\text{post} : \forall\alpha_1\alpha_2.(\alpha_1 \rightarrow \alpha_2) \times \alpha_1 \times \alpha_2 \rightarrow \text{prop}$$

These declarations are admissible by 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 [52], we have transcribed Objective Caml’s library implementation of finite sets, represented as balanced binary search trees, into our programming language.

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 $\rightarrow$ bool, whose specification could be written as follows:

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

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 $\rightarrow$ bool, such that the latter decides the former.

In a full-scale programming language, our balanced binary search tree implementation would be a function, parameterized over the type “elt”, the function “<”, the relation “<”, and the ordering axioms for “<”, and the function “<”.

We assume that an abstract type of sets of elements, written “set elt”, 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, this would be provided by a (logical-level) standard library.

7.2 Definitions

Figure 10 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 “$x$”, “$t$”, “$r$”, “$h$”, “$y$” are inferred. The types of the function “elements” and of the 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” [58, 3, 55] and “generalized algebraic data types” [59] 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 11 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
let rec member (x, t)  
where bst (t)  
returns u where b = true ⇔ x ∈ elements (u)  
= match t with  
| Empty → false  
| Node (y, l, r, _) →  
if x = y then  
true  
else if x < y then  
member (x, l)  
else  
member (x, r)  
end  

Figure 11. Membership in a binary search tree

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

Figure 13. A sample client of the iterator abstraction

“x” is a member of the set implemented by the tree “i”. 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 over binary search trees. Their expressive power surpasses that of “fold” (§7.5), yet their specification is simpler.

The implementation appears in Figure 12. 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.

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

The function “iterator” creates an iterator “i” out of a tree “t”, and satisfies the postcondition “ok(i)∧elements(t) ≡ remaining(i)”, where ≡ 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 “next”, when applied to an iterator “i”, returns either nothing or a pair of a new iterator “i’” and an element “x”. The precondition describes how these values are related.

Figure 13 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 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 “cardinal” function performs a loop, expressed as an internal recursive function, with an integer accumulator n. 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 “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 definition “is invariant” summarizes the required invariant. Its parameters are “inv”, the client’s invariant, which itself is parameterized over and accumulator and a set of remaining elements; “s₀”, a set of remaining elements; “s₁”, the initial set of all elements; the client function “f”; and an accumulator “a” etc. The definition states, in short, that: the set of remaining elements is a subset of the initial set of all elements; the current accumulator and the current set of remaining elements satisfy the client’s invariant; and, 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.

The function “fold” is parameterized over two ghost variables, namely the client invariant “inv” and a set “s₀” of remaining el-
predicate is_invariant (inv, s₀, s₁, f, accu) =
  s₀ ⊆ s₁ ∧ inv (accu, s₀) ∧
  ∀ s x s' accu' .
  s ⊆ s₀ ∧ s = s' ∪ {x} ∧ x ∉ s' ∧ inv (accu, s) ⇒
  pre (f) (accu, x) ∧
  post (f) (accu, x) (accu') ⇒ inv (accu', s')

let rec fold (logic s₀, logic inv, accu, t, f) where is_invariant (inv, s₀, elements (t), f, accu) ∧ bst (t) returns accu' where inv (accu', s₀) elements (t) =
match t with
| Empty → accu
| Node (x, l, r, ω) →
  let accu = fold (s₀, inv, accu, l, f) in
  let accu = f (accu, x) in
  fold (s₀ \ elements (l) \ {x}, inv, accu, r, f)
end

predicate cardinal inv (t) =
λ (accu, s). accu + ∥s∥ = ∥elements (t)∥

let cardinal (t) returns x where x = ∥elements (t)∥ =
fold (elements (t), cardinal inv (t), 0, t, λ (accu, x), accu + 1).

Figure 14. Folding over binary search trees

8. Related work
The roots of our work lie in Hoare logic [24, 30]. Extensions of Hoare logic with support for recursive, higher-order procedures were heavily studied in the late 1970's and early 1980's [12, 5, 15, 26, 27]. In particular, the issue of completeness received a lot of attention after Clarke [12] 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 [15] 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 [19, 22, 20, 21, 46, 6]. 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 Figures 6–9. In comparison with the method used in ESC/Java [23], 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 e₁ is made the first component of a sequence, as in "let e₂/false = e₁ in e₂", then second component e₂ is checked under the assumption false. As a result, all of the the proof obligations found within e₂ are vacuously satisfied. This can be sound only if e₂ is never executed, which is the case under call-by-value evaluation, but not under call-by-name evaluation. 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 [53] 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 [43]. More recent implementations [2, 8, 45] embed Scott's LCF within some form of higher-order logic. In a somewhat similar vein, Longley and Pollack [40] embed the functional core of Standard ML, via a fully abstract denotational semantics, into higher-order logic.

Our approach is more 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 [35, 29], the Cover translator [1], or Honda and Yoshida's logic of higher-order functions [31], 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.

ESC/Haskell [60] 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 [56] can be used as a programming language, in which programs are both developed and proved correct. The Cmpcert certified compiler [37] 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 [55] 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 [47, 46] 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 [7, 28, 60, 46] 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 [61, 58] and refinement types [16] 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 [59] 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 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 [59] 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 [50, 49].

Going beyond indexed types, several programming languages offer full dependent types [3, 10, 54, 57]. 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:

- publishing a usable prototype implementation, equipped 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 formulæ, together with suitable conversion rules for exploiting such equations; indeed, we, and other authors [46], have noticed that such an extension would subsume generalized algebraic data types [59];
- 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.

References
