Even Camels Fall into Coma: Flexible Barriers for Deductive Verification of OCaml Programs

Beatriz Rosas[1] and Paul Patault[2]

[1] NOVA University Lisbon, NOVA LINCS
[2] Paris-Saclay University, Laboratoire Méthodes Formelles, Inria

Abstract. In deductive verification, writing precise specifications remains a bottleneck: contracts often require verbose boilerplate that duplicates implementation. This paper introduces a novel verification workflow for Cameleer, a deductive verification tool for OCaml. Our approach targets Coma, an intermediate verification language that provides explicit control over abstraction barriers. We use this mechanism to support *in-depth specifications*, which can be attached to pattern-matching rather than only to function boundaries. This pipeline translates annotated OCaml programs into Coma, which are then verified by the Why3 backend. We demonstrate, on a set of verified case studies, that this approach reduces the number of specification lines by 25%.

Keywords: Deductive verification · OCaml · Gospel · Cameleer · Coma.

1 Introduction

Over the past decades, software systems have become increasingly integrated into everyday life, in domains where small failures can have significant consequences. Ensuring the correctness of such systems is therefore a central concern in software engineering. Deductive verification addresses this challenge by providing mathematical guarantees that a program satisfies a given formal specification [7]. In contrast to testing and simulation techniques, which can only demonstrate the presence of errors, formal verification aims to establish correctness properties for all possible executions.

Multiple frameworks have been developed to make deductive verification practical. One of them is Cameleer [19], which enables mostly-automated deductive verification of OCaml programs annotated with the specification language Gospel. This tool translates OCaml+Gospel programs into WhyML, which can then be verified using the Why3 framework [9] that supports automated provers, such as Alt-Ergo [4], CVC5 [2], and Z3 [5], and interactive provers, including Rocq [11] and Isabelle [18].

Although effective, the traditional specification approach of Cameleer/Why3 is not as flexible as we might want it. In particular, since specifications are written at function boundaries, they must describe the function's behavior across all possible execution paths. As a result, these contracts may become larger than the implementation itself, partially duplicating its structure, making them difficult to write and maintain. To address these limitations, we present an alternative workflow within Cameleer, based on Coma [16], a recently proposed intermediate verification language. Coma allows specifications to be mixed with executable code and provides explicit control

over *abstraction barriers*, the notion of separation between implementation and specification. Code located above a barrier is visible to callers and verified at every call site, while code below the barrier is hidden from callers and verified only once at its definition site. This mechanism allows programmers to define different levels of abstraction for various parts of a function, depending on the verification needs. We use this mechanism to support *in-depth specifications* for OCaml+Gospel programs. Instead of placing all specifications at the entry of a function, specifications may be attached to pattern-matching branches. This allows properties to be stated closer to the program fragments they describe, reducing duplication between code and specification.

In the remainder of the paper, we first motivate our approach through a Skew Heap example (Sec. 2). Then, we introduce the source and target languages of our translation (Sec. 3), followed by the proposed translation from annotated OCaml programs to Coma (Sec. 4). Next, we evaluate our approach on a set of verified case studies, comparing the number of lines of specification required by traditional top-level specifications and by in-depth specifications (Sec. 5). Finally, we discuss related work on deductive verification (Sec. 6) and conclude by outlining directions for future work (Sec. 7). The source code[3] and all the examples[4] are available online.

2 Motivating Example

To motivate our approach, we consider the data structure Skew Heap [22] that implements a priority queue using binary trees. The central operation of this data structure is `merge`, which combines two priority queues into one. The first line of the Gospel specification gives the name `r` to the result, and the remaining lines state pre- and postconditions, written with the `requires` and `ensures` keywords, respectively. This contract states the correctness properties of `merge`: a well-formedness heap invariant is preserved, the result contains exactly the elements of the two arguments, and its size is equal to the sum of their sizes. We assume given the predicate `heap`, and the logical functions `size` and `occ`, the number of occurrences of an element in a tree.

```
type tree = Empty | Node of tree * int * tree

let rec merge t1 t2 =
  match t1, t2 with
  | Empty, _ → t2
  | _, Empty → t1
  | Node (l1, x1, r1), Node (l2, x2, r2) →
      if x1 ⩽ x2 then Node (merge r1 t2, x1, l1)
      else Node (merge t1 r2, x2, l2)
(*@ r = merge t1 t2
    requires heap t1 && heap t2
    ensures  heap r
    ensures  ∀e. occ e r = occ e t1 + occ e t2
    ensures  size r = size t1 + size t2 *)
```

[3] Online at https://github.com/ocaml-gospel/cameleer/tree/coma.

[4] Online at https://beatrizspr9.github.io/inforum2026/ and archived with Zenodo at https://doi.org/10.5281/zenodo.20717441.

The previous function is particularly important since most update operations on skew heaps (e.g., insertion and removal) can be expressed as thin wrappers around it. Consequently, their specifications are not independent from the one of `merge`, but correspond to particular instances of it. For example, insertion can be implemented by merging the current heap with a singleton heap containing the new element.

```
let add x t = merge (Node (Empty, x, Empty)) t
(*@ r = add x t
    requires heap t
    ensures  heap r
    ensures  ∀e. e ≠ x → occ e r = occ e t
    ensures  occ x r = occ x t + 1
    ensures  size r = size t + 1 *)
```

In traditional deductive verification tools, one must give a specification to `add`, since the client of this function is verified using its contract. Ideally, such wrappers should instead be traversed by callers, allowing a call to `add` to be treated as a special case of `merge` with a singleton heap. In this setting, the verification condition (VC) may rely directly on the specification of `merge` avoiding duplication at the definition of `add`. This is where our approach using Coma comes into play, as it allows us to move the abstraction barrier from the function boundary into the function body. We can therefore hide recursive computations behind local specifications, while exposing non-recursive wrapper functions to callers. In particular, if we completely remove the specification of `add`, the resulting VC of the client code translated to Coma relies directly on the specification of `merge`.

```
let main t = add 42 t
(*@ r = main t
    requires heap t
    ensures  heap r *)
```

```
goal vc_main :
 ∀t. heap t →
  let t' = Node Empty 42 Empty in
  (heap t' && heap t) ∧
  (∀r. heap r ∧
   (∀e. occ e r = occ e t' + occ e t) ∧
   (size r = size t' + size t) →
    heap r)
```

The first part of the conjunction corresponds to the precondition of `merge`, while the other corresponds to its postcondition, both instantiated with the singleton heap `t'`.

The same idea applies inside the code of `merge`: removing the abstraction barrier from the two first branches reduces the proof effort for a caller with an empty tree.

```
let rec merge t1 t2 =
  match t1, t2 with
  | Empty, _ → t2
  | _, Empty → t1
  | Node (l1, x1, r1), Node (l2, x2, r2)
      (*@ requires heap t1 && heap t2
          ensures  heap result
          ensures  ∀e. occ e result = occ e t1 + occ e t2
          ensures  size result = size t1 + size t2 *)
      → ...
```

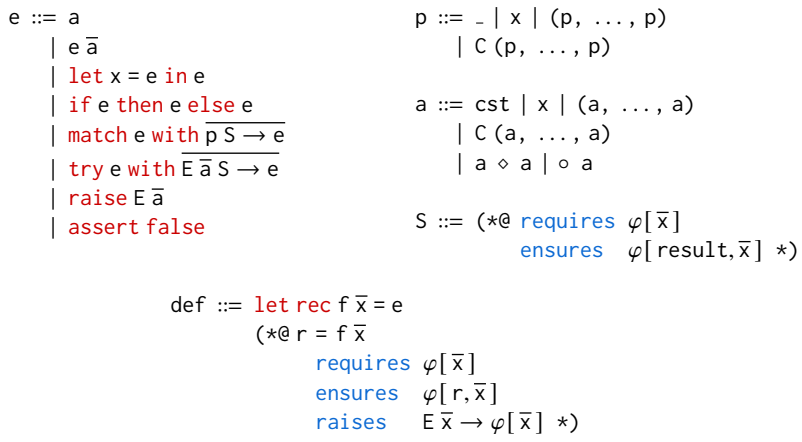


Fig. 1. Syntax of OCaml+Gospel subset.

3 The Languages At Both Ends

Gospel-Annotated OCaml. For the sake of simplicity—but without loss of generality—we assume that the source OCaml program is in A-normal form (ANF) [10]. This representation is commonly used as a first step for the *continuation-passing style* (CPS) translation. In ANF, expressions are divided into *atomic expressions*, which can be evaluated directly (e.g., variables and constants), and *complex expressions*, that require additional computation. Complex expressions must either be bound to a variable through a `let` construct or appear in tail position. This normalization makes the evaluation order explicit, contrary to the OCaml compiler which let it unspecified. The following example computes the height of a binary tree, written in ANF.

```
let rec height_anf t =
  match t with
  | Empty → 0
  | Node (l, _, r) →
      let hl = height_anf l in
      let hr = height_anf r in
      let m  = max hl hr in
      m + 1
(*@ r = height_anf t
    ensures r ⩾ 0 *)
```

Figure 1 presents the first-order, purely functional subset of OCaml considered in this work together with the Gospel annotations supported by our translation. For simplicity and clarity, type annotations are omitted. The body of a function is an expression, denoted `e`, which can be a value, an application of an expression to a list of atoms, a local variable definition, an `if` statement, a pattern-matching construction, an exception handler, an exception throw, or an assertion-failure expression. Patterns can

```
e ::= c ā c̄                                   π ::= (x: τ)⃗ (k: π → ⊥)⃗
    | let h π = e in e
    | assert φ ; e                             c ::= h | fun π → e
    | hide e

                    handler ::= let rec h π = e
```

Fig. 2. Syntax of Coma.

bind variables at an arbitrary depth of nesting in their related action. Atoms, denoted a, are pure terms made of constants, variables, constructor applications, and binary (and unary) total operators application (e.g., +, %, <). The specification is written in the Gospel logical language (that can be understood as a standard first-order logic, denoted φ) which is given inside structured comments `(*@ ... *)`, attached to function definitions or pattern analysis, and are preprocessed by the OCaml compiler.

Coma. The target language of our translation is Coma [16], an intermediate verification language implemented on top of Why3 [9]. Unlike traditional contract-based approaches, where specifications are attached only at function boundaries, Coma's specifications are assertions mixed with executable code. A central feature of Coma is its support for explicit abstraction barriers, which syntactically separate interface-level reasoning from implementation-level verification. The first part, located above a barrier, is visible to callers and verified at every call site (just the same as a standard precondition). The other part, hidden below a barrier, is invisible to callers and verified only once at the definition site.

Coma programs are written in *continuation-passing style*, where definitions never return but call a continuation to give back control. The syntax is presented in Fig. 2. A Coma program consists of a collection of *handlers*, the language construct used to represent executable computations. A handler can receive data parameters and continuation parameters. Their bodies are expressions, denoted e, which is either a call to a continuation (named or anonymous), a local continuation definition, a blocking assertion or an abstraction barrier, denoted `hide`, guarding the subsequent expression. Note that continuations always have the return type $\bot$ reflecting the fact that invoking a continuation transfers control and does not return to the caller. By making continuations explicit, control flow becomes more uniform and no special constructs are needed for pre- and postconditions. Preconditions are expressed as a combination of an in-code assertion followed by a barrier, while postconditions are encodable as preconditions of continuations. The following example illustrates how pre- and postconditions are written.

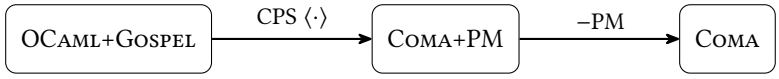
```
let rem x y =
    x - (x / y) * y
(*@ r = rem x y
      requires y > 0 ∧ x ⩾ 0
      ensures  x = (x / y) * y + r
      ensures  0 ⩽ r < y
 *)
```

```
let rem x y ret =
  let out r =
    assert x = (x / y) * y + r ;
    assert 0 ⩽ r < y ;
    hide ret r in
  assert y > 0 ∧ x ⩾ 0 ;
  hide out (x - (x / y) * y)
```

On the left, the program is written in OCaml+Gospel , and its corresponding Coma translation is shown on the right. It computes the remainder of the Euclidean division of `x` by `y`. On the Coma side, the result is given to the continuation `ret` through the wrapper `out`. This computation requires `y` to be different from zero, and the postconditions are written as preconditions of the wrapper. This wrapper is defined above the abstraction barrier at the last line of the definition of `rem`, which is required in order to make postconditions visible to clients of the handler. If a postcondition was placed below an abstraction barrier, it would become part of the hidden implementation and could no longer be used during verification at call sites. Moreover, it enables naming the result.

4 From Gospel-Annotated OCaml To Coma

Our translation proceeds in two stages. First, we CPS-translate the source program into Coma extended with pattern-matching, using the algorithm by Kennedy [13]. Then, we compile nested patterns into nested case analysis using the algorithm by Maranget et al. [15], where we erase the simple case analysis by declaring Coma destructors. The overall compilation schema is illustrated below.



4.1 Specification Preserving CPS-Translation

The language Coma+PM can easily be defined as a syntactic extension of the definition from Fig. 2 with

$$\mathtt{e} ::= \ldots \mid \mathtt{match}\ \mathtt{a}\ \mathtt{with}\ \overline{\mathtt{p} \longrightarrow \mathtt{e}}$$

where the patterns `p` are the same as the ones in Fig. 1. Notice that the specification annotation `S` disappears.

The translation from ANF to CPS introduces continuation parameters to every Coma+PM definition, which represent the possible destinations of control once the function has produced a result. The continuation parameter of the normal termination receives the computed result, while additional continuations may be used to model exceptional control flow (see Sec. 4.3). For example, the CPS-translation of `height_anf` is as follows:

```
let rec height_cps t ret =
  match t with
  | Empty → ret 0
  | Node (l, _, r) →
      let k3 m  = ret (m + 1) in
      let k2 hr = max_cps hl hr k3 in
      let k1 hl = height_cps r k2 in
      height_cps l k1
```

$$
\begin{aligned}
\langle\ \cdot \mid \cdot\ \rangle &: \alpha \to (\alpha \to \bot) \to \bot \\
\langle\ \mathsf{a} \mid \kappa\ \rangle &= \kappa\ \mathsf{a} \\
\langle\ \mathsf{e}\ \overline{\mathsf{a}} \mid \kappa\ \rangle &= \langle\ \mathsf{e} \mid \mathsf{fun}\ \mathsf{f} \to \mathsf{f}\ \overline{\mathsf{a}}\ \kappa\ \rangle \\
\langle\ \mathsf{let}\ \mathsf{x} = \mathsf{e}_1\ \mathsf{in}\ \mathsf{e}_2 \mid \kappa\ \rangle &= \mathsf{let}\ \mathsf{j}\ \mathsf{x} = \langle\ \mathsf{e}_2 \mid \kappa\ \rangle\ \mathsf{in}\ \langle\ \mathsf{e}_1 \mid \mathsf{j}\ \rangle \\
\langle\ \mathsf{if}\ \mathsf{e}_1\ \mathsf{then}\ \mathsf{e}_2\ \mathsf{else}\ \mathsf{e}_3 \mid \kappa\ \rangle &= \langle\ \mathsf{e}_1 \mid \mathsf{fun}\ \mathsf{z} \to \mathsf{let}\ \mathsf{j}\ \mathsf{x} = \kappa\ \mathsf{x}\ \mathsf{in} \\
&\qquad\qquad \mathsf{if}\ \mathsf{z}\ \mathsf{then}\ \langle\ \mathsf{e}_2 \mid \mathsf{j}\ \rangle\ \mathsf{else}\ \langle\ \mathsf{e}_3 \mid \mathsf{j}\ \rangle\ \rangle \\
\langle\ \mathsf{match}\ \mathsf{e}_1\ \mathsf{with}\ \overline{\mathsf{p}\ \mathsf{S} \to \mathsf{e}} \mid \kappa\ \rangle &= \langle\ \mathsf{e}_1 \mid \mathsf{fun}\ \mathsf{z} \to \mathsf{let}\ \mathsf{j}\ \mathsf{x} = \kappa\ \mathsf{x}\ \mathsf{in} \\
&\qquad\qquad \mathsf{match}\ \mathsf{z}\ \mathsf{with} \\
&\qquad\qquad |_i\ \mathsf{p}_i \to \mathsf{let}\ \mathsf{j'}\ \mathsf{result} = \\
&\qquad\qquad\qquad\qquad \mathsf{assert}\ \mathsf{ens}(\mathsf{S}_i)\ ; \\
&\qquad\qquad\qquad\qquad \mathsf{hide}\ \mathsf{j}\ \mathsf{result}\ \mathsf{in} \\
&\qquad\qquad\qquad\quad \mathsf{assert}\ \mathsf{req}(\mathsf{S}_i)\ ; \\
&\qquad\qquad\qquad\quad \mathsf{hide}\ \langle\ \mathsf{e}_i \mid \mathsf{j'}\ \rangle\ \rangle \\
\langle\ \mathsf{assert}\ \mathsf{false} \mid \kappa\ \rangle &= \mathsf{fail}
\end{aligned}
$$

Fig. 3. CPS-translation.

The continuation `ret` represents what is executed once `height_cps` has produced its result. In the `Empty` branch, the value `0` is immediately passed to this continuation, since no further computation is required. In the `Node` branch, the intermediate computations previously expressed using `let` bindings become explicit continuations: `k1` receives the height of the left subtree, `k2` receives the height of the right subtree, and `k3` receives their maximum value before passing the final height to `ret`.

The translation we follow is presented in Fig. 3, but is simplified due to space constraints[5]. From its two arguments, one source program to translate and its continuation κ, the translation produces a Coma+PM program. The base case is reached when we encounter an atom: the continuation applies. For the expression application, we must CPS-translate it and change the continuation to the function application. The local definition construct computes e_1 with a new defined continuation `j` that translates e_2 with continuation κ. For the `if` statement, we CPS-translate the Boolean and select the corresponding continuation depending on its value. The pattern-matching is the most complex case since it allows *in-depth* specification. First, we evaluate the scrutinee, and then we use the pattern-matching expression of our extended Coma+PM. Then, for each branch, we redefine the continuation if and only if there is an `ensures` clause in `S`; and same for the `requires` clause, we add it as a precondition of the action (if present). Finally, the assertion-failure expression leads to a call to the handler `fail` from the Coma standard library.

We can easily add optimizations. When continuation κ is a handler name (instead of a context), we can avoid to introduce of variable `j` in the cases for `if` and `match`. Moreover, when the condition of an `if` statement (or the scrutinee of a pattern-matching) is an atom, we do not need an intermediate variable to store the result of the computation. The rules are extended with a rule of the following form:

$$\langle\ \mathsf{if}\ \mathsf{a}\ \mathsf{then}\ \mathsf{e}_2\ \mathsf{else}\ \mathsf{e}_3 \mid \mathsf{k}\ \rangle = \mathsf{if}\ \mathsf{a}\ \mathsf{then}\ \langle\ \mathsf{e}_2 \mid \mathsf{k}\ \rangle\ \mathsf{else}\ \langle\ \mathsf{e}_3 \mid \mathsf{k}\ \rangle$$

[5] Details are available in the technical report https://hal.science/hal-05670171.

4.2 Pattern-Matching Compilation

After the CPS-translation, we compile nested patterns into nested match-with. In other words, we want to trade p for p_0—where p_0 allows only 0-depth patterns.

$$\begin{aligned} p &::= \_ \mid x \mid (p, \ldots, p) \mid C\ (p, \ldots, p) \\ p_0 &::= v \mid (v, \ldots, v) \mid C\ (v, \ldots, v) \qquad \text{with } v ::= \_ \mid x \end{aligned}$$

We use a standard pattern-matching compilation algorithm [14]. A match expression is represented as a *clause matrix*, whose rows correspond to branches and whose columns correspond to subterms of the scrutinee. The algorithm recursively considers the left-most column. If the column contains only variables or wildcards, it is eliminated by introducing the corresponding bindings. Otherwise, the algorithm produces one sub-matrix for each constructor appearing in that column and proceed recursively on each submatrix. Tuple patterns are treated uniformly as constructor patterns. A detailed description of the full algorithm is given in the technical report. Applying this transformation to the `merge` function, results in the following Coma+PM handler.

```
let rec merge t1 t2 =
  match t1, t2 with
  | Empty, _ → t2
  | _, Empty → t1
  | Node (l1, x1, r1),
    Node (l2, x2, r2)
    → ...
```

```
let rec merge t1 t2 k =
  match t1 with
  | Node (l1, x1, r1) →
     (match t2 with
      | Node (l2, x2, r2) → ⟨ ... | k ⟩
      | Empty → k t1)
  | Empty →
     (match t2 with
      | Empty → k t2
      | _ → k t2)
  | _ → (match t2 with Empty → k t1)
```

After this step, the remaining shallow matches must be encoded explicitly, since Coma does not provide a native pattern-matching construct. We use declared destructors: generated handlers that take one data parameter, the scrutinee, and a collection of continuations representing each branch of the match. Calling a destructor transfers control to one of these continuations, depending on the scrutinee constructor. This pattern-matching elimination transformation is written $[\![\cdot]\!]$.

$$[\![\ \texttt{match a with } \overline{\texttt{p} \to \texttt{e}}\ ]\!] = \texttt{destruct\_}i\ \texttt{a}\ \overline{(\texttt{fun}\ \texttt{p}_{args} \to [\![\ \texttt{e}\ ]\!])}$$

Here, `destruct_`i is a fresh destructor. Each branch is translated into a continuation parameter whose body is the recursive translation of the corresponding expression. Continuation parameters correspond to the variables bounded by the pattern, denoted p_{args}. For example, $(x)_{args} = x$, $(\_)_{args} = \emptyset$, and $(\texttt{Node(l,\_,r)})_{args} = \texttt{l r}$. Since patterns are restricted to 0-depth, this extraction is straightforward. Moreover, each declared destructor must be annotated with preconditions that capture the choice made in the case analysis. These preconditions are obtained by conjoining the branch condition with the negation of all preceding branches. They guarantee that every action is proved under the hypothesis that the scrutinee is of the form of its related pattern. They also guarantee mutual exclusivity between continuation branches.

For example, consider the outer match on `t1` in the translated version of `merge`. We generate a fresh destructor `destruct_merge1` with three continuations, corresponding to the three branches of this match. Each continuation is annotated with a precondition, represented as a formula inside angle brackets.

```
val destruct_merge1 (t: tree)
        (node (l: tree) (x: int) (r: tree) {t = Node (l, x, r)})
        (empty          {t = Empty && ∀lxr. t ≠ Node (l, x, r)})
        (default        {t ≠ Empty && ∀lxr. t ≠ Node (l, x, r)})
```

Following the same principle for the other pattern-matchings over `t2`, destructors `destruct_merge2`, 3, and 4 are generated and the resulting full translation is

```
let rec merge t1 t2 k =
  destruct_merge1 t1
    (fun l1 x1 r1 →
        destruct_merge2 t2 (fun l2 x2 r2 → ⟨ ... | k ⟩)
                           (→ k t2))
    (→ destruct_merge3 t2 (→ k t2)
                          (→ k t2))
    (→ destruct_merge4 t2 (→ k t1))
```

The pattern-matching constructs were completely removed and replaced by calls to the respective generated destructors. Following this mechanism, we can simply encode pattern-matching in Coma without introducing any new language construct.

4.3 Exceptions

Thanks to CPS, representing exceptions is straightforward. The translation requires only the introduction of additional continuation parameters to represent exceptional control flow, without introducing new proof obligations when compared to the standard translation. The precondition of each exceptional continuation is obtained from the corresponding Gospel `raises` clause. The following example illustrates it, where the function `div` is translated into a Coma handler with two continuations: one for normal termination, and one for the exceptional case. Raising an exception is therefore just a matter of invoking the corresponding exceptional continuation.

```
let div x y =
  if y = 0 then raise (DbZ x)
           else x / y
(*@ r = div x y
    requires x ⩾ 0 ∧ y ⩾ 0
    ensures  x ⩾ r     * y
    ensures  x < (r+1) * y
    raises   DbZ _ → y = 0 *)
```

```
let div x y (ret: int → ⊥)
            (exnDbZ: int → ⊥)
= let ret' r =
    assert x ⩾ r     * y ;
    assert x < (r+1) * y ;
    hide ret r in
  let exnDbZ' r =
    assert y = 0 ; hide exnDbZ r in
  assert x ⩾ 0 ∧ y ⩾ 0 ;
  hide if y = 0 then exnDbZ' x
                else ret' (x / y)
```

On the other hand, a `try-with` expression is handled by locally redefining the continuations associated with the caught exceptions. Consequently, exceptions handled

Example	LoC	LoS	LoS*	LoS†	Time	Time*	Time†
Binary Search Tree	87	70	58	58	1	1.08	1.61
Red Black Trees	60	135	127	108	1	0.98	–
Leftist Heaps	40	73	56	43	1	1.03	2.03
Pairing Heaps	34	73	56	52	1	1.09	1.24
Skew Heaps	25	43	25	21	1	1.01	1.41
Same Fringe	22	12	–	10	1	–	1.12

Table 1. Results of our evaluation. Column LoC indicates the lines of code of the verified program. Columns LoS, LoS*, and LoS† indicate the lines of specifications with *top-level*, *smart*, and *depth-most* specifications. Column Time is the relative time reference for the full verification of the code with *top-level specifications*. Columns Time*, and Time† indicate the verification time factor of *smart*, and *depth-most* specifications compared to the reference time.

inside the `with` block do not appear in the exceptional interface of the enclosing function, while unhandled exceptions continue to be propagated through the exceptional continuations. The translation of exceptions is discussed in more detail in the technical report.

5 Case Studies

We evaluate our approach on a collection of case studies derived from Why3's gallery[6]. The goal is to assess whether in-depth specifications reduce the number of specification lines, while preserving the ability to verify the same functional properties as the original implementations. We also measure the verification time of each benchmark to determine the impact of the specification writing style. These measurements were performed on a machine with an Intel Core i7-13800H (13th Gen) and 16 GB of RAM, running Fedora Linux 43. Verification times were measured using `hyperfine` [20] and the `replay` command of Why3 1.8.2, averaging 10 runs per benchmark.

The examples, written in WhyML, are reimplemented in OCaml and annotated with Gospel specifications. In addition, the programs must be rewritten into ANF. For each benchmark, we write three versions. The first version follows a *traditional approach*, where specifications are attached at function boundaries. The second version uses *depth-most specifications* (denoted †), where abstraction barriers are moved as deeply as possible into the function body. It isolates recursive computations behind local specifications, and entirely removes the specification from non-recursive functions. The third version uses *smart specifications* (denoted ⋆) which is a mix of the two previous approaches in order to trade off between the reduction in the number of specification lines and the verification time. Then, all the versions are compiled into Coma by our tool before being verified by Why3.

[6] Online at https://toccata.gitlabpages.inria.fr/toccata/gallery/why3.en.html.

Table 1 summarizes the results of our evaluation. As expected, the number of specification lines decreases in every benchmark when only using the depth-most specifications approach. The largest reductions, reaching up to 50%, are observed in the Skew Heaps and Leftist Heaps examples. These benchmarks contain many non-recursive functions that can be directly exposed to callers. However, in all of these cases, the verification time becomes significantly higher than the original top-level specifications. This is a consequence of exposing functions to callers: their proof is no longer done at definition site but at each call site, which may end up in larger verification conditions that repeat some proofs. This is where the third approach makes sense: we reintroduce some specifications that are not necessary for correctness but helpful to the prover. It results in verification times comparable to the original ones, while still achieving a significant reduction in specification size.

Note that the depth-most approach for Red Black Trees could not be verified within a reasonable time, hence the missing time entry (marked "–"). Also note that Same Fringe does not show results for *smart specification*: in this case, since the example is too simple, there is nothing to do better than top-level version.

To further analyze the impact of depth-most specifications on verification time, we implemented a client of the Skew Heaps example that performs successive calls to `add`. When the number of insertions grows (i.e., up to sixteen successive calls), the impact of inlining becomes much more evident. Using the top-level contract of `add`, the client is verified in 0.25 seconds, whereas exposing and fully inlining its implementation increases the verification time up to 9.5 seconds.

More generally, the results show the major impact of the placement of abstraction barriers on the specification size, its redundancy with the code, and the verification time required by the provers.

6 Related Work

The verification of OCaml programs has been explored through several tools, each offering different degrees of automation and expressiveness. CFML [3], for instance, provides a deductive framework for verifying programs written in a subset of OCaml. Although it targets the same source language as Cameleer, CFML adopts a fundamentally different verification strategy. Top-level definitions are translated into a corresponding characteristic formula, a higher-order logical formula that gives a sound and complete description of the semantics of a program fragment. These formulas are then proved interactively using Rocq [11]. The framework has been used to verify CPS programs, including list concatenation, and higher-order iterators. Despite its expressiveness, CFML relies heavily on user interaction since users are required to guide the proof process, limiting automation when compared to SMT-based approaches.

For reasoning about concurrent programs, Iris [12] has emerged as a higher-order concurrent Separation Logic formalized in Rocq. Iris is language-agnostic, making it a versatile tool for reasoning about concurrent programs. It extends traditional Separation Logic with additional operators and reasoning rules, enabling more expressive reasoning about shared state. To encode Iris semantics for OCaml, two distinct projects have been developed: Osiris [21] and Zoo [1]. Osiris targets a larger frag-

ment of OCaml and employs a monadic interpreter that uses a tree-based representation for computations. Zoo, by contrast, focuses on verifying fine-grained concurrent algorithms, and adopts a small-step operational semantics.

Outside the deductive verification setting, Gospel has also been used by Ortac [8], a runtime assertion checking tool for OCaml. Ortac identifies the executable subset of Gospel formulas from annotated module interfaces and translates them into OCaml expressions. These generated wrappers treat the implementation as a black box by checking preconditions before a function call, verifying postconditions and invariants after normal returns, and checking exceptional postconditions when exceptions are raised. Ortac also provides a modular plugin architecture, enabling different uses of the generated code, including traditional runtime assertion checking, monitoring, fuzzing, and model-based testing.

Recent work on Creusot [6] has also explored Coma as a backend for deductive verification of Rust programs [17]. Creusot follows a similar approach to Cameleer, where annotated Rust programs are translated into WhyML and processed by Why3. In the Coma-based pipeline, Rust functions are translated into independent Coma modules, from which verification conditions are generated and discharged through Why3. A key motivation for this extension is that specifications for small closures are often longer than the closures themselves, despite merely restating their behavior. The Coma backend addresses this issue by inferring closure specifications from their bodies, reducing the need for explicit annotations while preserving modular verification. The reported evaluation shows substantial reductions in specification size across several benchmarks, with verification times remaining comparable overall. Paskevich, Patault, and Filliâtre [16] also report that replacing Creusot's backend with Coma often improves verification-condition generation time, with speedups above 50% for a significant fraction of the benchmarks.

7 Conclusion & Future Work

We presented a new verification workflow for Cameleer based on Coma, an intermediate verification language with explicit abstraction barriers. The proposed pipeline translates annotated OCaml programs into Coma, and verifies them using Why3's backend. We extended Gospel's specification layer, enabling *in-depth specifications* inside pattern-matching, allowing specifications to be placed closer to the program fragments they describe. We evaluated this approach on a set of verified case studies by comparing traditional top-level specifications with in-depth specifications. The results show that moving abstraction barriers inside function bodies can reduce the number of specification lines by 25% average, decreasing the annotation burden while preserving a deductive verification workflow for OCaml programs.

As future work, we plan to extend our pipeline to support a larger fragment of OCaml. In particular, we intend to handle mutable state, which is essential for verifying more realistic programs. We are also interested in exploring the continuation-passing style nature of Coma to study the translation of higher-order functions and the inference of specifications for such programs.

Acknowledgments. We would like to thank Mário Pereira, Andrei Paskevich, and Jean-Christophe Filliâtre for their invaluable assistance and the numerous constructive discussions about Cameleer, Coma, and Why3 that made this work possible. This work was partly supported by Agence Nationale de la Re-cherche (ANR) grant ANR-22-CE48-0013-01 (GOSPEL) and NOVA LINCS (UID/04516/2025) with the financial support of FCT.IP (https://doi.org/10.54499/UID/04516/2025). The authors acknowledge the support provided by Fondation Inria, within the framework of the OCaml Software Foundation, to UNINOVA under a Donation Protocol for the Development of Scientific Research.

References

1. Clément Allain and Gabriel Scherer. Zoo: A framework for the verification of concurrent OCaml 5 programs using separation logic. *Proc. ACM Program. Lang.*, 10(POPL), January 2026.
2. Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. CVC5: A versatile and industrial-strength smt solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 415–442, Cham, 2022. Springer International Publishing.
3. Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, pages 418–430, 2011.
4. Sylvain Conchon, Albin Coquereau, Mohamed Iguernlala, and Alain Mebsout. Alt-ergo. In *SMT Workshop: International Workshop on Satisfiability Modulo Theories*, 2018.
5. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer.
6. Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. Creusot: A foundry for the deductive verification of Rust programs. In *International Conference on Formal Engineering Methods*, pages 90–105. Springer, 2022.
7. Jean-Christophe Filliâtre. Deductive software verification. *International Journal on Software Tools for Technology Transfer*, 13(5):397–403, October 2011.
8. Jean-Christophe Filliâtre and Clément Pascutto. Ortac: Runtime assertion checking for OCaml (tool paper). In Lu Feng and Dana Fisman, editors, *Runtime Verification*, pages 244–253, Cham, 2021. Springer International Publishing.
9. Jean-Christophe Filliâtre and Andrei Paskevich. Why3—where programs meet provers. In *European symposium on programming*, pages 125–128. Springer, 2013.
10. Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, page 237–247, New York, NY, USA, 1993. Association for Computing Machinery.
11. Hugo Herbelin, Pierre-Marie Pédrot, coqbot, Gaëtan Gilbert, Maxime Dénès, letouzey, Emilio Jesús Gallego Arias, Matthieu Sozeau, Théo Zimmermann, Enrico Tassi, Jean-Christophe Filliatre, Pierre Roux, Guillaume Melquiond, Jason Gross, Arnaud Spiwack, barras, Pierre Boutillier, Vincent Laporte, Jim Fehrle, Stéphane Glondu, Yves Bertot, Jasper Hugunin, Clément Pit-Claudel, Ali Caglayan, forestjulien, Pierre Courtieu, Yann Leray,

Frédéric Besson, Pierre Rousselin, and MSoegtropIMC. rocq-prover/rocq: Rocq 9.2.0, March 2026.
12. Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018.
13. Andrew Kennedy. Compiling with continuations, continued. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, pages 177–190, 2007.
14. Fabrice Le Fessant and Luc Maranget. Optimizing pattern matching. *ACM SIGPLAN Notices*, 36(10):26–37, 2001.
15. Luc Maranget. Compiling pattern matching to good decision trees. In *Proceedings of the 2008 ACM SIGPLAN workshop on ML*, pages 35–46, 2008.
16. Andrei Paskevich, Paul Patault, and Jean-Christophe Filliâtre. Coma, an intermediate verification language with explicit abstraction barriers. *ACM Trans. Program. Lang. Syst.*, May 2026. Just Accepted.
17. Paul Patault, Arnaud Golfouse, and Xavier Denis. Remonter les barrières pour ouvrir une clôture. In *JFLA 2025-36es Journées Francophones des Langages Applicatifs*, 2025.
18. Lawrence C. Paulson. Isabelle: The next 700 theorem provers, 2000.
19. Mário Pereira and António Ravara. Cameleer: A deductive verification tool for OCaml. In *International Conference on Computer Aided Verification*, pages 677–689. Springer, 2021.
20. David Peter. hyperfine: A command-line benchmarking tool. https://github.com/sharkdp/hyperfine.
21. Remy Seassau, Irene Yoon, Jean-Marie Madiot, and François Pottier. Formal semantics and program logics for a fragment of OCaml. *Proc. ACM Program. Lang.*, 9(ICFP), August 2025.
22. Daniel Dominic Sleator and Robert Endre Tarjan. Self adjusting heaps. *SIAM J. Comput.*, 15(1):52–69, February 1986.